ON MATCHING BINARY TO SOURCE CODE

Arash Shahkar

A thesis
in
The Department
of
Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
in Information Systems Security at
Concordia University
Montréal, Québec, Canada

March 2016

© Arash Shahkar, 2016

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Arash Shahkar**

Entitled: **On Matching Binary to Source Code**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Jia Yuan Yu ______________________ Chair

Dr. Lingyu Wang ______________________ Examiner

Dr. Zhenhua Zhu ______________________ External Examiner

Dr. Mohammad Mannan ______________________ Supervisor

Approved ______________________________
Chair of Department or Graduate Program Director

____________ 2016 ______________________________

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

On Matching Binary to Source Code

Arash Shahkar

Reverse engineering of executable binary programs has diverse applications in computer security and forensics, and often involves identifying parts of code that are reused from third party software projects. Identification of code clones by comparing and fingerprinting low-level binaries has been explored in various pieces of work as an effective approach for accelerating the reverse engineering process.

Binary clone detection across different environments and computing platforms bears significant challenges, and reasoning about sequences of low-level machine instructions is a tedious and time consuming process. Because of these reasons, the ability of matching reused functions to their source code is highly advantageous, despite being rarely explored to date.

In this thesis, we systematically assess the feasibility of automatic binary to source matching to aid the reverse engineering process. We highlight the challenges, elaborate on the shortcomings of existing proposals, and design a new approach that is targeted at addressing the challenges while delivering more extensive and detailed results in a fully automated fashion. By evaluating our approach, we show that it is generally capable of uniquely matching over 50% of reused functions in a binary to their source code in a source database with over 500,000 functions, while narrowing down over 75% of reused functions to at most five candidates in most cases. Finally, we investigate and discuss the limitations and provide directions for future work.

Contents

List of Figures viii

List of Tables ix

Code Listings x

1 Introduction 1

1.1 Motivation . . . 2

1.2 Thesis Statement . . . 3

1.3 Contributions . . . 3

1.4 Outline . . . 5

2 Background 6

2.1 Software Compilation and Build Process . . . 6

2.1.1 High-level Source Code . . . 7

2.1.2 Abstract Syntax Tree . . . 9

2.1.3 Intermediate Representation . . . 10

2.1.4 Control Flow Graph . . . 10

2.1.5 Compiler Optimizations . . . 11

2.1.6 Machine Code . . . 15

2.1.7 Linking . . . 15

2.2 Binary to Source Matching . . . 17

2.2.1 Automatic Compilation . . . 17

2.2.2 Automatic Parsing 19

3 Related Work **20**

3.1 Binary to Source Comparison 20

3.2 Binary Decompilation 22

3.2.1 Decompilation as an Alternative 24

3.2.2 Decompilation as a Complementary Approach 25

3.3 High-Level Information Extraction from Binaries 25

3.4 Source Code Analysis 26

3.5 Miscellaneous 27

4 CodeBin Overview **29**

4.1 Assumptions 29

4.2 Comparison of Source Code and Binaries 30

4.3 Function Properties 31

4.3.1 Function Calls 31

4.3.2 Standard Library and API calls 31

4.3.3 Number of Function Arguments 32

4.3.4 Complexity of Control Flow 33

4.3.5 Strings and Constants 35

4.4 Annotated Call Graphs 37

4.5 Using ACG Patterns as Search Queries 40

5 Implementation **42**

5.1 Challenges 42

5.1.1 Macros and Header Files 42

5.1.2 Statically Linked Libraries 44

5.1.3 Function Inlining 46

5.1.4 Thunk Functions 47

5.1.5 Variadic Functions 48

5.2 Source Code Processing 49
5.2.1 Preprocessing and Parsing 49
5.2.2 Source Processor Architecture 51
5.3 Binary File Processing 53
5.3.1 Extracting Number of Arguments 53
5.3.2 ACG Pattern Extraction 53
5.4 Graph Database 55
5.4.1 Subgraph Search 55
5.4.2 Query Results Analysis 57
5.5 User Interface 57

6 Evaluation **61**
6.1 Methodology 62
6.1.1 Test Scenario 62
6.1.2 Pattern Filtering 63
6.1.3 Result Collection and Verification 64
6.2 Evaluation Results 64
6.3 No Reuse 67
6.4 Different Compilation Settings 68
6.5 Source Base and Indexing Performance 70

7 Discussion **73**
7.1 Limitations 73
7.1.1 Custom Preprocessor Macros 73
7.1.2 Orphan Functions 75
7.1.3 Inaccurate Feature Extraction 76
7.1.4 Similar Source Candidates 76
7.1.5 C++ Support 77
7.2 CodeBin as a Security Tool 79
7.3 Directions for Future Work 79

8 Conclusion **81**

Bibliography **89**

List of Figures

1 Sample abstract syntax tree 9
2 Control flow graph created from source code 11
3 The effect of LLVM optimizations on CFG 13
4 The effect of MSVC optimizations on CFG 14
5 Cyclomatic complexity of four different hypothetical CFGs 34
6 Cyclomatic complexity of source and binary functions without compiler optimizations 35
7 Cyclomatic complexity of source and binary functions with compiler optimizations 36
8 Sample partial annotated call graph 38
9 Overall design of CodeBin 40
10 The effect of static linking on binary ACGs 45
11 Thunk functions 48
12 The architecture of CodeBin source code processor. 50
13 User interface: Indexing source code 58
14 User interface: Inspecing ACGs 59
15 User interface: Viewing matching results 60
16 User interface: Viewing source code 60

List of Tables

1 Complementary information on Figure 8 39
2 Results of evaluating CodeBin in real-world scenarios. 65
3 CodeBin results in no-reuse cases. 67
4 Effect of different compilation settings on CodeBin's performance. . . 69
5 CodeBin test dataset, parsing and indexing performance. 72

Code Listings

1 Sample C source code. 8
2 Part of x86 assembly code for findPrimeSpeed function 16
3 Cypher query for the ACG pattern in Figure 8 56
4 Different implementations for ROTATE in OpenSSL. 74
5 Similar functions in Sqlite 77

Chapter 1

Introduction

Analysis and reverse engineering of executable binaries has extensive applications in various fields such as computer security and forensics [76]. Common security applications of reverse engineering include analysis of potential malware samples or inspecting commercial off-the-shelf software; and are almost always performed on binaries alone. Reverse engineering of programs in binary form is often considered a mostly manual and time consuming process that cannot be efficiently applied to large corpuses.

However, companies such as security firms and anti-virus vendors often need to analyze thousands of unknown binaries a day, emphasizing on the need for fast and automated binary analysis and reverse engineering methods. To this end, there has been several efforts in designing and developing reliable methods for partial or full automation of different steps of reverse engineering and binary analysis.

Code reuse is referred to the process of copying part of an existing computer program code in another piece of software with no or minimal modifications. Code reuse allows developers to implement parts of a program functionality by relying on previously written and tested code, effectively reducing the time needed for software development and debugging.

Previous research has shown that code reuse is a very common practice in all sorts of computer programs [17, 44, 60, 63, 75], including free software, commercial off-the-shelf solutions and malware, all of which are typical targets of reverse engineering.

In the process of reverse engineering a binary program, it is often desirable to quickly identify reused code fragments, sometimes referred to as clones. Reliable detection of clones allows reverse engineers to save time by skipping over the fragments for which the functionality is known, and focusing on parts of the program that drive its main functionality.

This thesis is the result of exploring identification and matching of reused portions of binary programs to their source code, a relatively new and less explored method in the area of clone-based reverse engineering.

1.1 Motivation

The source code of a computer program is usually written in a high-level programming language and is occasionally accompanied by descriptive comments. Therefore, understanding the functionality of a piece of software by reading its source code is much easier and less error-prone compared to analyzing its machine-level instructions. On the other hand, as will be discussed later in Chapter 2, while huge repositories of open source code is accessible for the public, creating large repositories of compiled binaries for such programs bears significant challenges. As a result, matching reused portions of binary programs to their source code would be an effective method in speeding up the reverse engineering process.

Previous efforts have been made on identifying reused code fragments by searching through repositories of open source programs [13, 39, 67]. However, research on binary to source code matching is still very scarce. All previous proposals in this area follow relatively simple and similar methods for matching executable binaries to source code, and have not been publicly evaluated beyond a few limited case studies. Also, to the best of our knowledge, there exists no systematic study on the feasibility and potential challenges of matching reused code fragments from binary programs to source code.

1.2 Thesis Statement

In summary, the objective of this thesis is to assess the feasibility of automatically comparing executable binary programs on the Intel x86 platform to a corpus of open/-known source code to detect code reuse and match binary code fragments to their respective source code. To reach this goal, we try to answer the following questions:

Question 1. What are the common features or aspects of a computer program that can be effectively extracted from both its source code and executable binaries in an automated fashion?

Question 2. How do compilation and build processes affect these aspects, and what are the challenges of automated binary to source comparison?

Question 3. Can we improve the existing solutions for binary to source matching by enriching the analyses with more reliable features and working around the challenges?

Question 4. To what extent can we use better binary to source matching techniques to detect code reuse and facilitate reverse engineering of real world binary programs?

1.3 Contributions

1. **Identification of Challenges.** We have explored the effects of the compilation and build process on various aspects of a program, and have shown why certain popular features that are commonly used in previous work for comparing source code [19, 45, 47, 69] or binary code fragments [30, 31, 32, 41, 46, 71] together cannot be used for comparing source code to binary code. We have also studied and hereby describe the challenges of automatically extracting features from large corpuses of open source code, as well as the technical complications of binary to source comparison.

2. **New Approach.** We explore the possibilities of improving current binary to source matching proposals by studying additional features that can be used for

comparing source code to binary code, and propose a new approach, CodeBin, which unlike previous proposals, tries to move from syntactic matching to extracting semantic features from source code and is capable of revealing code reuse with much more detail.

3. **Implementation.** We have implemented CodeBin and created a system that is capable of automating the matching of reused binary functions to their source code on arbitrary binary programs and code bases. Our implementation can automatically parse and extract features from arbitrary code bases, create searchable indexes of source code features, extract relevant features from disassembled binaries, match binary functions to previously indexed source code, and generally reveal the source code of a majority of reused functions in an executable binary.

4. **Evaluation.** We have evaluated CodeBin by simulating real world reverse engineering scenarios using existing open source code. We present the results of code reuse detection through binary to source matching, and assess the scalability of our approach. To this end, we have indexed millions of lines of code from 31 popular real world software projects, reused portions of the previously indexed code in 12 binary programs, and used our prototype implementation to detect and match reused binary functions to source functions. In summary, our implementation is generally capable of uniquely matching over 60% of reused functions to their source code, requiring approximately one second for indexing each 1,000 lines of code and less than 30 minutes for analyzing relatively large disassembled binary files on commodity hardware. We have also tested our system in 3 cases where no previously indexed code is reused, and have investigated the results in all cases. We present sample cases and describe the reasons why the capabilities of our approach is undermined in certain circumstances, and also discuss the possibilities for future work.

1.4 Outline

The rest of this thesis is organized as follows. Chapter 2 covers some necessary background on how source code is translated into machine level executable binary programs, as well as the challenges of automatically matching binary programs to source code. In Chapter 3, we review related work in the field of reverse engineering and software analysis, and discuss previous proposals for binary to source matching and their shortcomings. Chapter 4 is dedicated to the formulation of our approach, including the results of studying additional common features that can be extracted from source code and binaries and our proposed method of creating searchable indices from extracted features as well as performing searches over indexed source code. We present the details of our prototype implementation in Chapter 5, and discuss certain aspects of it that are incorporated to address several technical challenges that are caused by the complexity of the compilation process. In chapter 6, we present the results of evaluating our approach in real world clone detection-based reverse engineering scenarios and our dataset. Chapter 7 includes the results of investigating the evaluation results, covers challenging cases with samples and potential opportunities for future work and improvements. Finally, we conclude in Chapter 8.

Chapter 2

Background

2.1 Software Compilation and Build Process

The build process, in general, is referred to the process of converting software source code written in one or several programming languages into one or several software artifacts that can be run on a computer. An important part of the build process is compilation, which transforms the source code usually written in a high-level programming language into another target computer language that is native to the platform on which the program is meant to run. The target code is then processed further to create target software artifacts.

The details of the build process highly varies between programming languages and target platforms. For instance, building a program written in C into a standalone executable binary for Intel x86 CPUs is completely different from building a Java program that is executed in a Java Virtual Machine (JVM) [54].

Our goal is to facilitate reverse engineering of binary programs. C is arguably the most popular programming language that is used to produce machine-executable binaries [78], and x86 is still the predominant computing platform. Due to the nature of our work, in the rest of this section we discuss the build process by focusing on C as the source code programming language and Intel x86 as the target platform.

2.1.1 High-level Source Code

A computer program is usually written in a high-level programming language such as C. These languages provide strong abstraction from the implementation details of a computer, and allow the programmer to express the logic of the program using high-level semantics.

We hereby discuss some related concepts in C programs using a simple example. This sample code, along with many others, were used in the early stages of this work to explore different ideas for binary to source matching.

Listing 1 shows a simple C program that receives an integer using a command line argument and computes the largest prime number that is smaller than the input integer using a pre-allocated array. The main logic of the program for finding the target prime number is encapsulated in the function $findPrimeSpeed$, and the target integer is passed to this function using its only argument, *limit*.

This program relies on three functions not defined in its source code, but imported from the C standard library: `malloc` for allocating space in memory, `printf` for writing in the standard output, and `atoi` for extracting an integer value from a character string. Functions, as well as structs and other components can be imported from other source files using the `#include` preprocessor directive.

As the first step towards creating an executable program from C source code, the code is preprocessed. Each C source file usually refer to several other source (header) files using `#include` directives, which instruct the preprocessor to simply replace the directive line with the contents of the included header file. Preprocessor macros are unique tokens defined by the programmer to replace arbitrary code, or to simply include or exclude certain parts of source code in combination with `#ifdef` and `#ifndef` directives. These macros shape an integral part of the C programming language and are commonly used for a variety of reasons, including controlling the build process and enclosing platform-dependent parts of code. In some projects such as OpenSSL, macros are heavily used to include different implementations of the same components, only one of which is eventually compiled. During the preprocessing step,

C compilers rewrite the source code based on a set of defined macros and form a concrete version of the code. As a result, a piece of C source code is likely to be incomplete without having a specific set of defined macros.

```c
#include <stdio.h>
#include <stdlib.h>

int findPrimeSpeed(int limit) {

    int *numbers = (int*) malloc(sizeof(int) * limit);
    int lastPrime;
    int n, x;

    for (n = 2; n < limit; n++) {
        numbers[n] = 0;
    }

    for (n = 2; n < limit; n++) {
        if (numbers[n] == 0) {
            lastPrime = n;
            for (x = 1; n * x < limit; x++) {
                numbers[n * x] = 1;
            }
        }
    }

    return lastPrime;

}

int main(int argc, char* argv[]) {
    printf("%d\n", findPrimeSpeed(atoi(argv[1])));
}
```

Listing 1: Sample C source code.

2.1.2 Abstract Syntax Tree

Once the code is preprocessed, a lexer parses the program's source code to convert its text into a parse tree or *concrete syntax tree*, which consists of all its tokens separated according to C syntax. The parse tree is then simply converted into an *abstract syntax tree* or AST, which is an immediate representation of source code based on its syntactic structure. AST differs subtlety from a parse tree as it does not represent all the details appearing in the real syntax, such as comments and spaces. However, each node in AST denotes a construct in the source code, and can represent anything from an operator to the name of a function argument. Obtaining an AST is therefore vital to any analysis of source code based on its semantics.

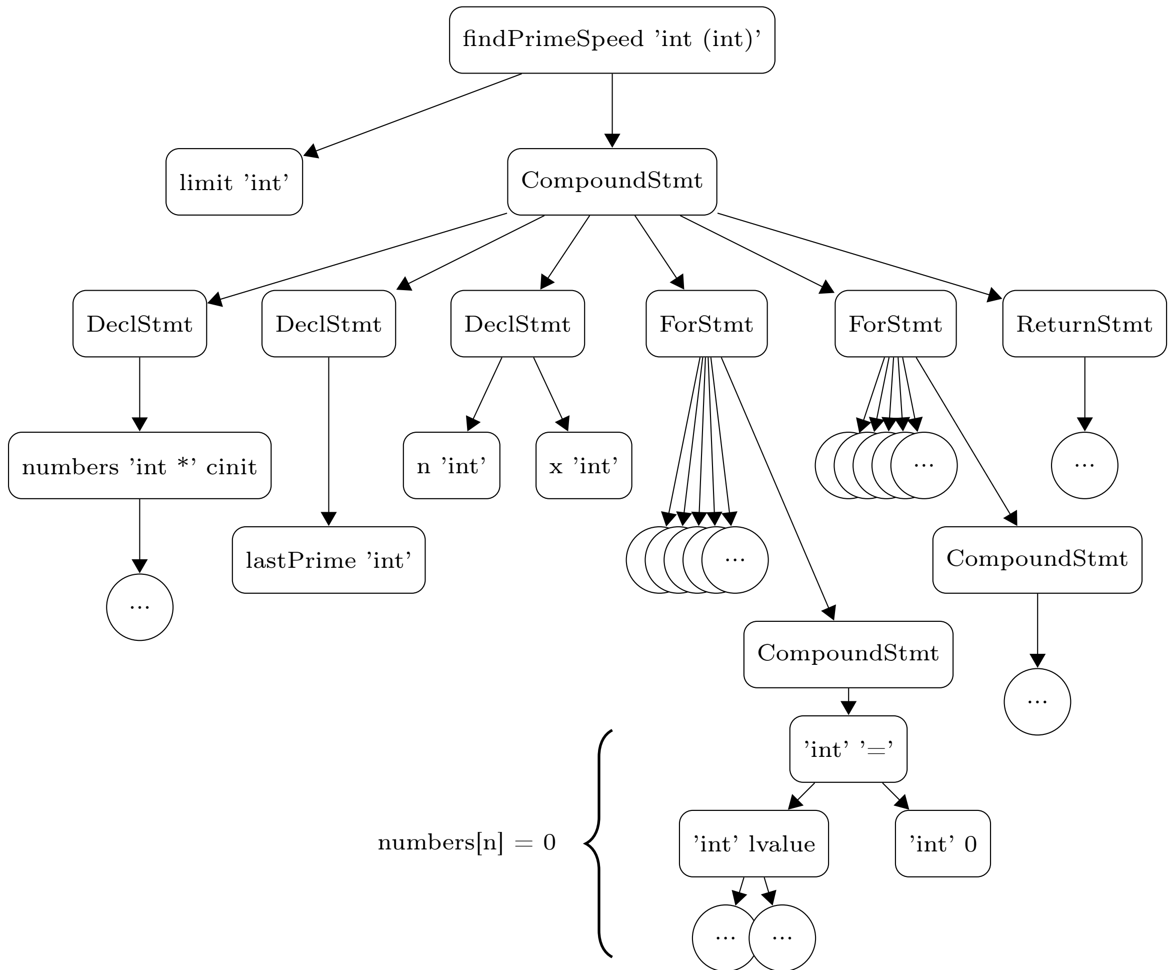

Figure 1: Abstract syntax tree for findPrimeSpeed function in Listing 1, created using Clang [48].

2.1.3 Intermediate Representation

A computer program can be represented in many different forms. Translating code from one language to another, as performed in compilation, requires analysis and synthesis, which are in turn tightly bound to the representation form of the program.

Compilers usually translate code into an intermediate representation (IR), also referred to as intermediate language (IL) [77]. Intermediate representations used in compilers are usually independent of both the source and target languages, allowing for creation of compilers that can be targeted for different platforms. Most syntheses and analyses are performed over this form of code.

For instance, the GCC compiler uses several different intermediate representations. These intermediate forms are internally used throughout the compilation process to simplify portability and cross-compilation. One of these IRs, GIMPLE [58], is a simple, SSA-based [20], three-address code represented as a tree that is mainly used for performing code-improving transformations, also known as optimizations. Another example is the LLVM IR [49], a strongly-typed RISC instruction set used as the only IR in the LLVM compiler infrastructure.

2.1.4 Control Flow Graph

A control flow graph (CFG) is a directed graph that represents all the possible flows of control during a program execution. CFGs are usually constructed individually for each function. In a CFG, nodes represent basic blocks and edges represent possible flows of control from one basic block to another, also referred to as jumps. A basic block is a list of instructions that always execute sequentially, starting at the first instruction and ending at the last.

A CFG can be created from any form of code, including source code (AST), its intermediate representations, or machine-level assembly code. As will be discussed later in this chapter, CFGs are an important form for representing a function or a program in general, and are commonly used in reverse engineering.

Figure 2 shows the control flow graph of the *findPrimeSpeed* function created by parsing the code into an AST and converting it to a CFG using Joern [83].

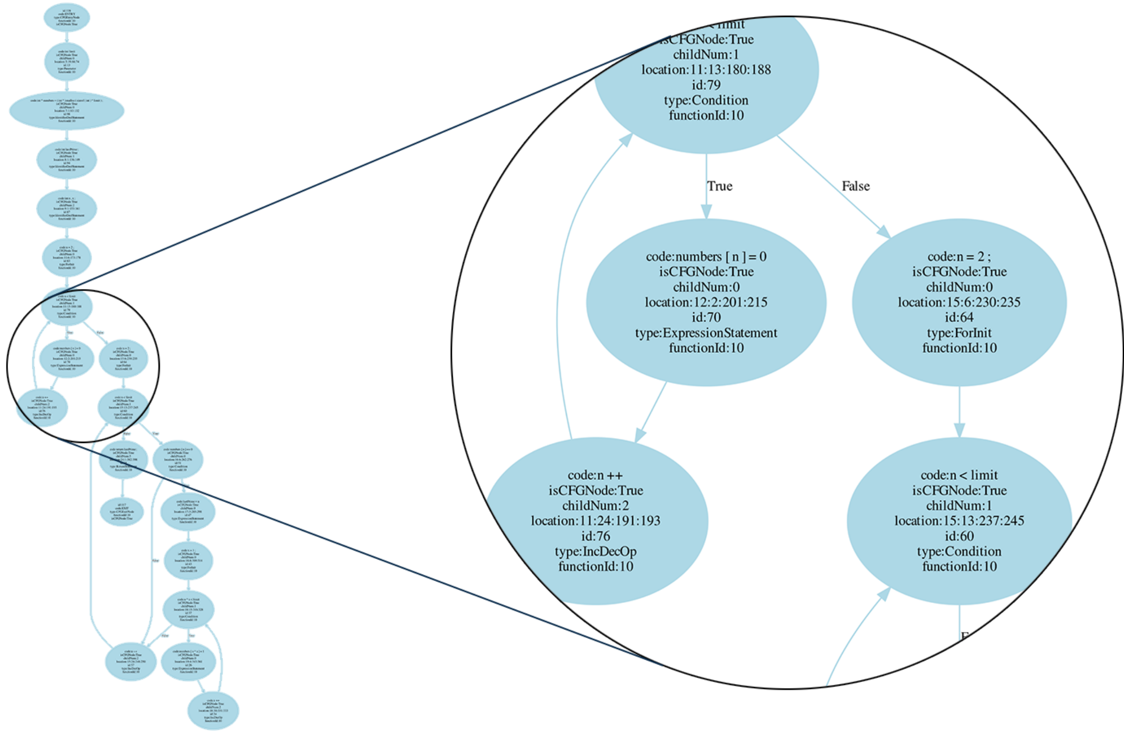

Figure 2: Control flow graph of findPrimeSpeed function as created from the original source code.

2.1.5 Compiler Optimizations

Compilers in general, and C compilers in particular, are capable of automatically performing a variety of code optimizations, i.e., transformations that minimize the time and/or memory required to execute the code without altering its semantics. These optimizations are typically implemented as a sequence of transformation passes, which are algorithms that take a program as input and produce an output program that is semantically equivalent, but syntactically different.

Although the target platform is a very important factor that affects code optimizations, a majority of such optimizations are intrinsically language and platform-independent [77]. On the other hand, to guarantee semantics preservation, the compiler should be able to perform several analyses such as data flow analysis and dependency analysis on the code, most of which are facilitated by intermediate languages. Based on these two reasons, most compiler optimizations are performed over the intermediate representation.

Since compiler optimizations are generally CPU and memory-intensive, compilers typically allow programmers to choose the level of optimization, an option that affects the time required for the compilation to finish, and how optimized the output is.

Compiler optimizations can potentially make significant changes to a piece of code, including considerable modifications of its control flow. The amount of changes introduced to a function as a result of compiler optimizations depends on several factors, including how optimal the original code is and what opportunities exist for the compiler to optimize it.

Figure 3 shows two control flow graphs for the *findPrimeSpeed* function. The CFG on the left side is derived from the LLVM IR that is obtained by directly translating the C source code, without performing any further analysis or optimization. The CFG on the right side is derived from the LLVM IR that is fully optimized, i.e., the output of the LLVM optimizer instructed to optimize the IR as much as possible. Similarly, Figure 4 shows the CFG of *findPrimeSpeed* as derived from x86 assembly produced using Microsoft Visual C compiler, without any optimization and with full optimization.

Two interesting observations can be made by comparing the LLVM IR CFGs (Figure 3) and the x86 assembly CFGs (Figure 4) to the source CFG (Figure 2):

1. **Language-independent nature of CFG.** Combining the sequential basic blocks of the source CFG yields a control flow graph that is *structurally* identical to the unoptimized CFGs obtained from LLVM IR or x86 assembly. In other words, the control flow graph seems to be language-independent: an abstract

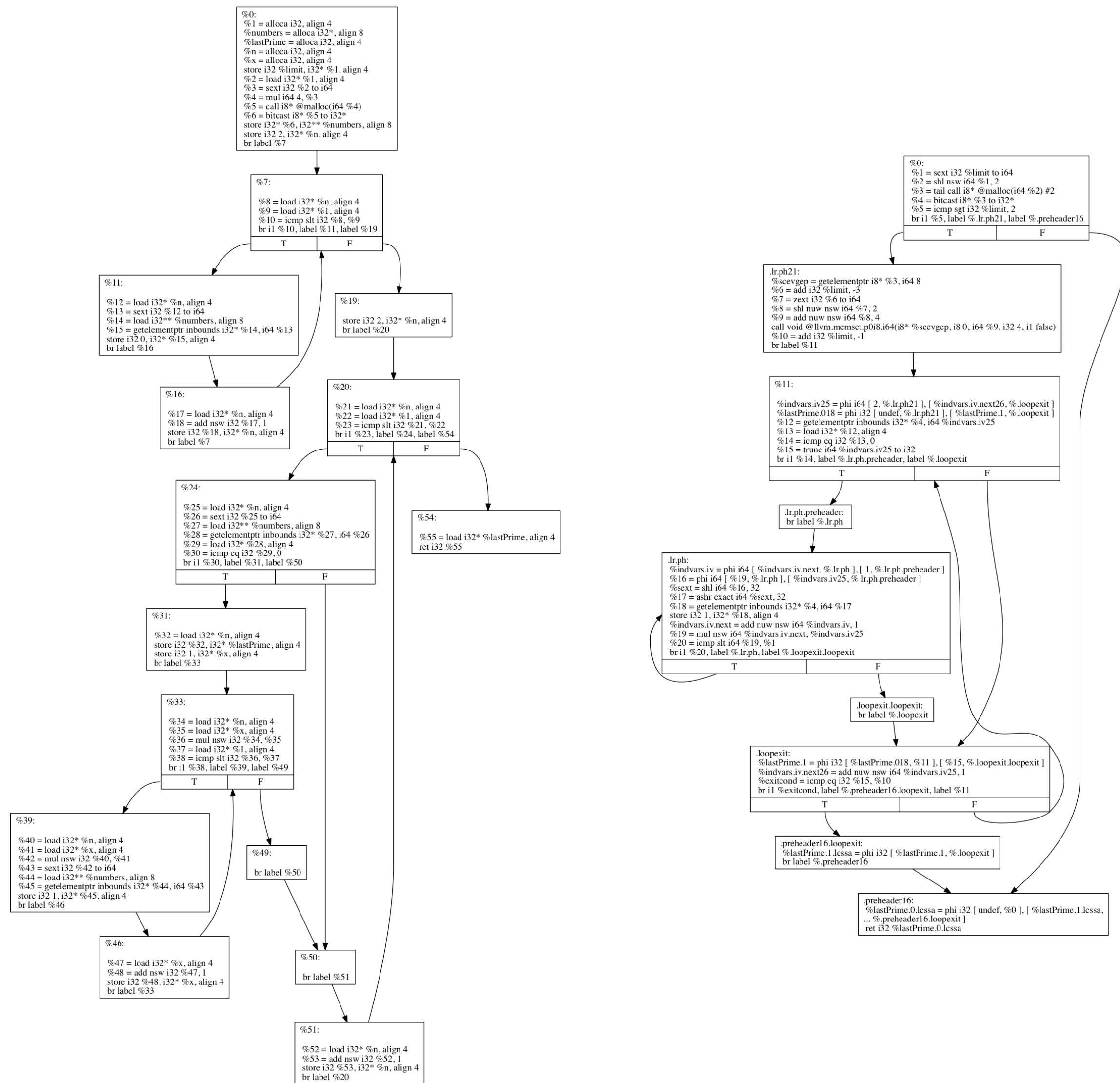

Figure 3: Control flow graph of findPrimeSpeed function as created from LLVM bytecode. Left: Without any optimization. Right: With full LLVM optimization.

feature that is capable of representing the flow of code regardless of what language it is written in. This is one of the key reasons why CFGs are an important form of representing a piece of code, as well as understanding it during reverse engineering, simply because its overall structure is generally not affected by the complexities of native, low-level machine languages such as the x86 instruction set.

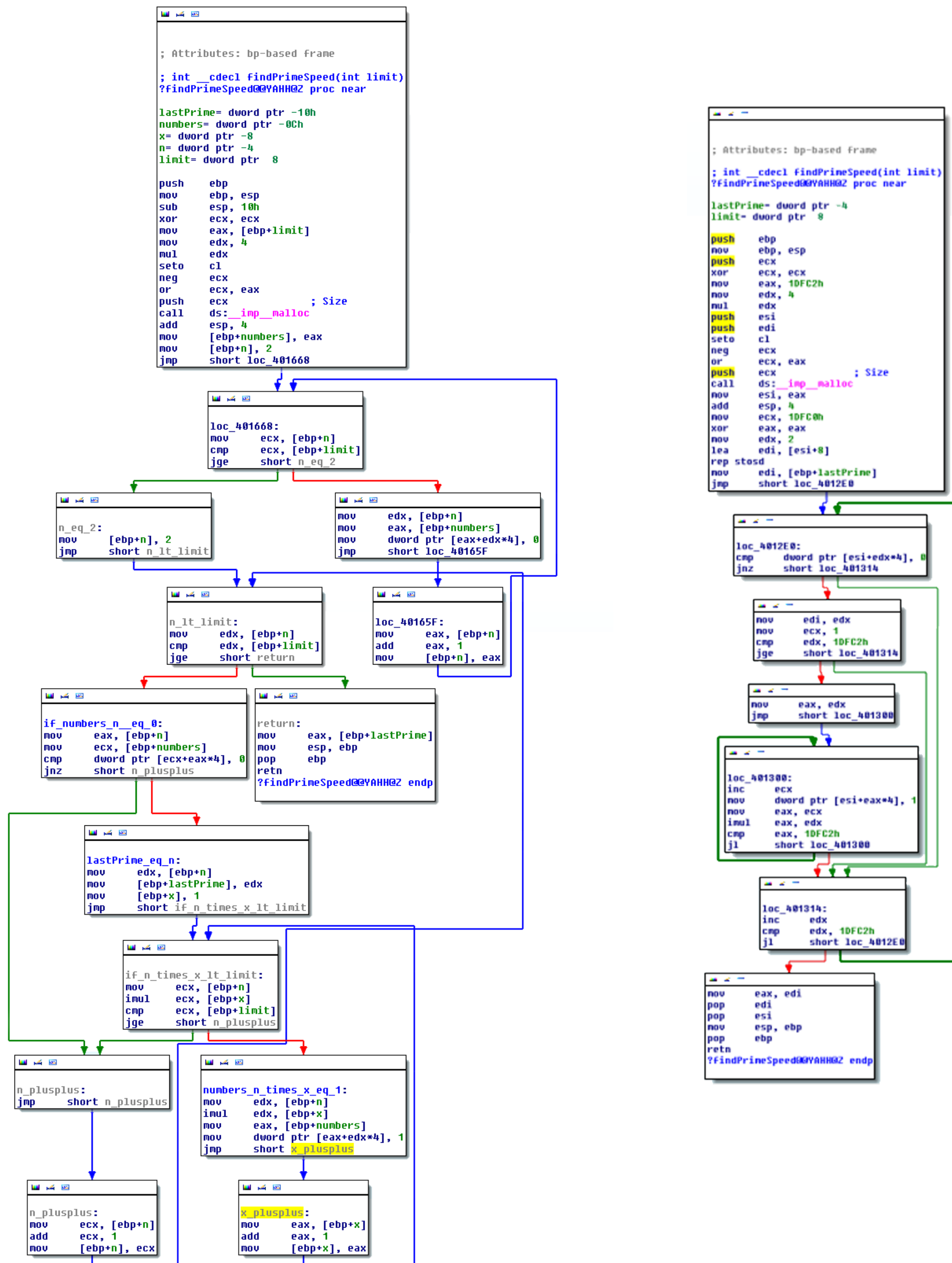

Figure 4: Control flow graph of findPrimeSpeed function as created from the assembly output of the Microsoft Visual C compiler. Left: Without any optimization. Right: With full (level 3) optimization.

2. **The effect of compiler optimizations on the CFG.** Note how compiler optimizations have significantly changed the control flow of a rather simple function. Generally, as a function becomes bigger with a more complex control flow, compilers are provided with more opportunities to transform it into a more optimal, but semantically equivalent code. Consequently, the chance of the CFG being changed as a result of these optimizations also increases. Therefore, while the structure of CFG seems to be language-independent, it cannot be effectively used to compare functions in source and target forms in the presence of an optimizing compiler.

2.1.6 Machine Code

Once architecture-independent analyses and optimizations are performed on the intermediate representation, the code is passed to a machine code generator: A compiler that translates IR code into native instructions for the target platform contained in an object file. In C and some other languages such as Fortran, compilation is done on a file-by-file basis: Each source file is translated into the intermediate and/or target language separately. Therefore, for our simple example in Listing 1, only one object file will be created.

The nature of the generated machine code relies heavily on the architecture of the target platform. Listing 2 contains part of the assembly code of *findPrimeSpeed* in the Intel syntax, as generated by Clang/LLVM [48] with full optimizations for the Intel x86 platform. The first part of the machine code related to the calling convention [55] is removed for brevity.

2.1.7 Linking

Object files are created per source file, contain relocatable machine code and are not directly executable. A linker is then responsible for linking various object files and libraries and creating the final executable binary. There are certain link-time

```
        mov     r14d, edi
        movsxd  r15, r14d
        lea     rdi, [4*r15]
        call    _malloc
        mov     rbx, rax
        cmp     r15d, 3
        jl      LBB0_6
        mov     rdi, rbx
        add     rdi, 8
        lea     eax, [r14 - 3]
        lea     rsi, [4*rax + 4]
        call    ___bzero
        dec     r14d
        mov     ecx, 2
        .align  4, 0x90
LBB0_2:
        cmp     dword ptr [rbx + 4*rcx], 0
        mov     edx, 1
        mov     rsi, rcx
        jne     LBB0_5
        .align  4, 0x90
LBB0_3:
        movsxd  rax, esi
        mov     dword ptr [rbx + 4*rax], 1
        inc     rdx
        mov     rsi, rdx
        imul    rsi, rcx
        cmp     rsi, r15
        jl      LBB0_3
        mov     eax, ecx
LBB0_5:
        mov     rdx, rcx
        inc     rdx
        cmp     ecx, r14d
        mov     rcx, rdx
        jne     LBB0_2
LBB0_6:
        add     rsp, 8
        pop     rbx
        pop     r14
        pop     r15
        pop     rbp
        ret
```

Listing 2: Part of x86 assembly code for findPrimeSpeed function

optimizations, mostly known as inter-procedural optimizations (IPO), that may be performed only during linking as the optimizer has the full picture of the program. Link-time optimizations may be performed on the intermediate representation or on the object files. In either case, the linker takes all the input files and creates a single executable file for each target.

Even the simplest pieces of code typically rely on a library, i.e., external code defined in sets of functions or procedures. For instance, our example relies on three library functions: *malloc*, *atoi* and *printf*. Libraries can be linked either statically or dynamically. Statically linked libraries are simply copied into the binary image, forming a relatively more portable executable. A dynamically linked library only has its symbol names included in the binary image and should be present in the system in which the binary is executed.

2.2 Binary to Source Matching

Having the essentials of the software build process explained briefly, we now discuss two opposing ideas for matching binaries to reused source code, and the reason we opted for the latter: Automatic compilation and automatic parsing.

2.2.1 Automatic Compilation

An idea for identifying reused source code in binary programs is to compile the source code to obtain a binary version, and to utilize binary clone detection techniques afterwards. We have explored this idea during the early stages of this project and have faced several significant obstacles. According to our observations, automatic compilation of an arbitrary piece of source code bears significant practical challenges. Here, we enumerate and explain some of the key obstacles for automatic compilation, which highlight the importance of the capability of directly comparing source code and binaries.

2.2.1.1 Various Build Configurations

As discussed before in Section 2.1.1, C code is preprocessed before parsing, a process that is heavily affected by preprocessor macros. An automatic compilation system faces a big challenge for obtaining a set of correct values for custom preprocessor macros. Usually, certain sets of values for these macros are included in a configuration script that comes with the codebase and is run before running the actual build script. However, different projects utilize different build systems, resulting in various methods for configuration and build. Therefore, it is difficult in practice to obtain a set of macros needed to compile a piece of code without any prior knowledge about the build system used. While modern build systems such as CMake [2] provide a cross-platform way of targeting multiple build environments and make the build process highly standardized, they are yet to be adopted by the majority of C/C++ code bases.

2.2.1.2 External Dependencies

Relying on external libraries for carrying out certain operations is a very common practice. These external dependencies are not necessarily included in the dependent projects and need to be downloaded, compiled in a compatible fashion and provided separately by the user. Automatic compilation requires a standardized system for retrieving and building these dependencies. These dependencies are usually either downloaded by build automation and dependency management scripts or documented to be read and installed by users. While standard dependency management systems are widely adopted by other languages [9, 35, 57], C/C++ projects have yet to embrace such dependency management systems, further hindering automatic compilation.

2.2.1.3 Cross-Compilation

At the time of writing this thesis, binary clone detection techniques are generally not reliable when applied on binaries compiled with very different configurations [31],

such as different compilers and different optimization levels (e.g., little optimization vs. heavy optimization), or on different platforms (e.g., x86 vs. ARM). On the other hand, there might be a large number of candidate projects over which the search will run. In this case, there needs to be a solution to automatically compile all the source code into binaries, ideally with different compilers and different levels of optimization. If the underlying platform of the future target binaries is not known, the automatic compilation step should also build the projects on different platforms to obtain a good quality set of binaries to match against.

2.2.2 Automatic Parsing

As a result of the challenges discussed above, we do not aim for compiling target source code into machine binaries as a first step towards source to binary comparison. Instead, we process the source code by parsing it and traversing the AST for extraction of key features that are later used for matching. Obviously, custom configuration macros are still an issue. However when only AST creation is considered, lacking knowledge about these macros results in a partially inaccurate process instead of blocking it completely, a problem that occurs when executable machine code is to be created. Similarly, the location of header files will also be a missing piece of information. In Chapter 5, we will explain our approach for obtaining ASTs in a fully automated fashion, without access to predefined custom macros or the location of header files.

Chapter 3

Related Work

In this chapter, we discuss previous work on reverse engineering of executable binaries, with a focus on high-level information recovery from program binaries and source code as well as assembly to source code comparison.

3.1 Binary to Source Comparison

To the best of our knowledge, the idea of reverse engineering through binary to source comparison is not explored much. There exist three tools and a few publications that focus on comparing binary programs to source code for various purposes including reverse engineering, all of which employ very similar preliminary techniques for comparison.

The oldest proposal, RE-Google [13, 50, 51], is a plugin for IDA Pro introduced in 2009. RE-Google is based on Google code search, a discontinued web API provided by Google that allowed third party applications to submit search queries against Google's open source code repository. RE-Google extracts constants, strings and imported APIs from disassembled binaries using IDA functionalities, and then searches for the extracted tokens to find matching strings in hosted source files. This plugin was left unusable once Google discontinued the code API.

The RE-Source framework and its BinSourcerer tool [66, 67] is an attempt to

recreate the functionalities originally provided by RE-Google using other online open source code repositories such as OpenHub [11]. BinSourcerer is also implemented as an IDA Pro plugin and follows a method very similar to RE-Google by converting strings and constants as well as imported APIs in binaries to searchable text tokens.

Methods proposed in RE-Google and RE-Source are indeed very similar, and both are based on syntactic string tokens and text-based searches. There are two major drawbacks of these proposals:

1. Both rely on online repositories for searching, limiting their capabilities in terms of source code analysis to what these online repositories expose in their APIs. This essentially prevents these proposals from being capable of fine-grained analysis, as online repositories APIs treat source code as text and only expose text-based search to third party applications [10]. A successful search using these tools returns a list of source files that contain the searched string token, each of which may be quite large and contain thousands of lines of code. The string tokens has an equal chance of being included in a comment and an actual piece of code, and may also be part of code written in any language, including the ones that are unlikely to be compiled into executable binaries [51].

2. One cannot use RE-Google or BinSourcerer to compare binaries to any arbitrary codebase, e.g., proprietary code that is not necessarily open source and is of interest for applications such as copyright infringement detection. This shortcoming may be addressed by creating a database of the non-open code accompanied by a searchable index as a secondary target for querying.

The binary analysis tool (BAT) [38, 39], introduced in 2013, is a generic lightweight tool for automated binary analysis with a focus on software license compliance. The approach adopted by BAT is also similar to that of RE-Google and BinSourcerer, as it also searches for text tokens extracted from binaries in publicly available source code. BAT is capable of extracting additional identifiers such as function and variable names from binary files, provided that they are attached to the binary. However, real-world

executable binaries are often stripped of such high-level information, and reasonable identifiers used by BAT are practically limited to strings.

It should be noted that despite their limitations, these tools may provide the reverse engineer with very useful information fairly quickly. For example, returning similar files all including implementations of cryptographic hash functions informs the reverse engineer that the binary file or function under analysis includes such a functionality. However, not all functions include distinctive strings and constants, a general factor that limits the potential of approaches that are purely based on syntactic tokens.

As will be shown later in this thesis, a piece of source code contains significantly more high-level information rather than just strings and constants, some of which can be effectively compared to binary files and functions for detecting reused portions of code.

Cabezas and Mooji [23] briefly discuss the possibility of utilizing context-based and partial hashes of control flow graphs for comparing source code to compiled binaries in a partially manual process. They do not however test, validate or provide evidence for the feasibility of this approach. Also, as we showed through an example in Section 2.1.5, CFGs change significantly once source code is transformed into binaries using an optimizing compiler.

3.2 Binary Decompilation

There has been several previous efforts on binary decompilation, which tries to generate equivalent code with high-level semantics from low-level machine binaries.

Historically, research on decompilation dates back to the 1960's [15]. However, modern decompilers have their roots in Cifuentes' PhD thesis in 1994 [27], where she introduced a structuring algorithm based on interval analysis [16]. Her proposed techniques is implemented in dcc [26], a decompiler for Intel 80286 / DOS binaries into C, which resorts back to outputting assembly in case of failure. The correctness

of dcc's output is not tested.

Another well-known decompiler, Boomerang [1], was created based on Van Emmerik's proposal [80] for using the Single Static Assignment (SSA) form for data flow components of a decompiler, including expression propagation, function signature identification and dead code elimination. Van Emmerik performed a case study of reverse engineering a single Windows program by using Boomerang with some manual analysis. However, other research efforts in this area have reported very few cases of successful decompilation using Boomerang [72].

HexRays Decompiler [6, 36] is the *de facto* industry standard compiler, available as a plugin for IDA Pro [7]. As of 2015, the latest version of HexRays is capable of decompiling both x86 and ARM binaries, providing full support for 32-bit and 64-bit binaries alike on both platforms. To the best of our knowledge, no other binary decompiler is capable of handling such a wide variety of executable binaries.

Phoenix [72] is another modern academic decompiler proposed by Schwartz et al. in 2013. Pheonix relies on BAP [22], a binary analysis platform that lifts x86 instructions into an intermediate language for easier analysis, and contains extensions such as TIE [52] for type recovery and other analyses. Phoenix employs semantics-preserving structural analysis to guarantee correctness and iterative control flow structuring to benefit from several opportunities for correct recovery of control flow that other decompilers reportedly miss [72]. Phoenix output is reportedly up to twice as more accurate as HexRays in terms of control flow correctness, but is unavailable for public use as of this writing.

Yakdan et al. proposed REcompile [81] in 2013, a decompiler that similar to dcc employs interval analysis for control flow structuring, but also uses a technique called node splitting to reduce the number of produced GOTO statements in the output. This technique reportedly has a downside of increasing the overall size of the decompilation output. DREAM [82] is another decompiler proposed by the same group in 2015, which also is focused on reducing the number of produced GOTO statements by using structuring algorithms that are not based on pattern matching,

a common method used in other decompilers. Neither of these two decompilers are available for public use as of the date if this writing.

In summary, previous research in the area of decompilation has highlighted significant challenges in correct recovery of types as well as control flow. For instance, experiments made by Schwartz et al. on Phoenix decompiler have shown several limitations and failures in terms of correct decompilation caused by floating-point operations, incorrect type recovery, inability to handle recursive structures and some calling conventions [72]. Also, the HexRays decompiler, which is the only usable and publicly available tool in this domain, does not perform type recovery [6] and is shown to be limited in terms of correct recovery of control flow [72, 82].

Despite its limitations, decompilation can be considered both as an alternative and a complementary approach when compared to binary to source matching.

3.2.1 Decompilation as an Alternative

In cases where there is no code reuse, decompilation output is very likely to be more usable compared to the results of any binary to source matching approach. Also, as will be shown later in Section 6.2, we have found that there are still cases where binary to source matching is likely to fail to provide useful results (see Section 7.1.2).

On the other hand, we argue that binary to source matching has a lot more potential in clone-based reverse engineering scenarios. Source code usually comes with many identifiers such as identifier names (functions, variables, structures, etc.) and comments that significantly facilitate understanding it, all of which are removed in a compiled binary in realistic settings. In these cases, correct matching from binary to source may provide a reverse engineer with more helpful results compared to correct decompilation.

3.2.2 Decompilation as a Complementary Approach

Decompilation can also be used as a complementary approach to binary to source matching. Due to limitations mentioned above, we have not tried to use a decompiler output in our work, except for recovering the number of arguments for a function in an optional fashion. However, one might try to apply several source-level clone detection techniques proposed in the literature [69] to benefit from some of the in-depth analyses decompilers perform for comparing binary functions to source functions.

3.3 High-Level Information Extraction from Binaries

There has been several research efforts on inference and extraction of high-level information such as variable types [29, 52], data structures [53, 68, 74] and object oriented entities [43, 70] from executable binaries using both static and dynamic analysis techniques. Some of these proposals achieve promising results in particular scenarios. However, we have not been able to use them in our work as they all have considerable limitations either in terms of relying on dynamic analysis and execution, focusing on very specific compilation settings, not supporting many realistic use cases, or simply not being available for evaluation.

Zhiqiang et al. developed REWARDS [53], an approach for automatic recovery of high-level data structures from binary code based on dynamic analysis and binary execution. REWARDS is evaluated on a subset of GNU coreutils suite, and achieves over 85% accuracy for data structures embedded in the segments that it looks into. TIE [52] is a similar approach by JongHyup et al. that combines both static and dynamic analysis to recover high-level type information. TIE is an attempt towards handling control flow and mitigating less than optimal coverage, both of which are significant limitations of approaches that are merely based on dynamic analysis. Based on an experiment on a subset of coreutils programs, TIE is reported to be 45% more accurate than RWARDS and HexRays decompiler. However, other work such as Phoenix [72] has shown considerable limitations of TIE in handling data

structures, and code with non-trivial binary instructions.

OBJDIGGER [43] is proposed by Jin et al. to recover C++ object instances, data members and methods using static analysis and symbolic execution. While it is shown to be able to recover classes and objects from a set of 5 small C++ programs, it does not support many C++ features such as virtual inheritance and is only targeted towards x86 binaries compiled by Microsoft Visual C++.

Prakash et al. proposed vfGuard [64], a system that is aimed at increasing the control flow integrity (CFI) protection for virtual function calls in C++ binaries. vfGuard statically analyzes x86 binaries compiled with MSVC to recover C++ semantics such as VTables.

There also exist a few old IDA Pro plugins for recovering C data structures [33] and C++ class hierarchies [73] using RTTI [87], but they all seem limited in terms of capabilities and are not actively developed in the public.

3.4 Source Code Analysis

Necula et al. have developed CIL [62], a robust high-level intermediate language that aids in analysis and transformations of C source code. CIL is both lower level than ASTs and higher level than regular compiler or reverse engineering intermediate representations, and allows for representation of C programs with fewer constructs and clean semantics. While it is extensively tested on various large C programs such as the Linux kernel, it needs to be run instead of the compiler driver to achieve correct results [61]. This basically means that one needs to modify the configure and make scripts that ship with the source code. As discussed earlier in Section 2.2.1.1, while this is not a limitation by any means, it makes automatic processing of arbitrary codebases infeasible due to customized build scripts.

SafeDispatch [40] is proposed by Jang et al. to protect C++ virtual calls from memory corruption attacks, a goal very similar to that of vfGuard [64]. However, SafeDispatch inserts runtime checks for protection by analyzing source code. This

system is implemented as a Clang++/LLVM extension, and needs to be run along with the compiler. Due to this requirement and similar to CIL [62], SafeDispatch cannot be automatically invoked on arbitrary C++ source bases.

Joern [83] is a tool developed by Yamaguchi et al. for parsing C programs and storing ASTs, CFGs and program dependence graphs in Neo4J [59] graph databases. Joern is used along with specific-purpose graph queries for detecting vulnerabilities in source code [84, 85, 86]. During our experiments, we have found that graphs created by Joern are not always reliable, specially in the presence of rather complex C functions or moderate to heavy use of custom preprocessor macros. As will be discussed later in Section 5.2, we adopt Clang [48], a mature open source modular compiler for parsing C source code.

3.5 Miscellaneous

Although not directly related, there are other pieces of work that make use of some of the key concepts we focus on in this thesis for a variety of purposes.

Lu et al. [56] propose a source-level simulation (SLS) system that annotates source code with binary-level information. In such a system, both source and binary versions of the code are available. The goal is to simulate the execution of the code, usually on an embedded system, while allowing the programmer to see how and by what extent specific parts of the code contribute to simulation metrics. The authors propose a hierarchical CFG matching technique between source and binary CFGs based on nested regions to limit the negative effect of compiler optimizations on SLS techniques.

In a recent paper [24], Caliskan-Islam et al. implement a system for performing authorship attribution on executable binaries. They use lexical and syntactic features extracted from source code and binary decompiler output to train a random forest classifier, and use the resulting machine learning model to de-anonymize the programmer of a binary program. Features include library function names, integers, AST node term frequency inverse document frequency (TFIDF) and average depth

of AST nodes among few others. The authors claim an accuracy of 51.6% for 600 programmers with fixed optimization levels on binaries. Although it is claimed that syntactic features such as AST node depth survive compilation, we have not been able to verify such a fact specially when multiple optimization levels, limitations of decompilers and hundreds of thousands of functions with potentially multiple programmers are to be considered.

Chapter 4

CodeBin Overview

We now outline our general approach for matching executable binaries to source code.

4.1 Assumptions

The underlying assumption is that the binary program under analysis may have used portions of one or more open source projects or other program for which the source code is available to the analyst, and identifying the reused code is a critical goal in the early stages of the reverse engineering process. Functions are usually considered as a unit of code reuse in similar work on binary clone detection. However, in some cases only part of a function might be reused. We do not aim at detecting partial function reuse in this work. Our main goal, therefore, is to improve the state of the art for binary to source matching by targeting individual binary functions instead of the entire executable. We do not aim at identifying all the functions in a piece of executable binary, but rather those that are reused and of which the source code is available.

We also do not consider obfuscated binaries, as de-obfuscation is considered as an earlier step in the reverse engineering process [37, 79].

4.2 Comparison of Source Code and Binaries

We introduce a new approach for identifying binary functions by searching through a repository of pre-processed source code. This approach aims at automatically finding matches between functions declared in different code bases and machine-level binary functions in arbitrary executable programs. At the core of CodeBin, we identify and carefully utilize certain features of source code that are preserved during the compilation and build processes, and are generally independent of the platform, compiler or the level of optimization. As a result, these features can also be extracted from binary files using particular methods, and be leveraged for finding similarities between source code and executable binaries at the function level without compiling the source code.

Due to the sophisticated transformations applied on source code by an optimizing compiler, we have found that the number of features that can be extracted from both plain source code and executable binaries and then directly used for comparison is rather small. Detailed properties of a binary function such as its CFG or machine instructions are often impossible to predict solely with access to its source code and without going into the compilation process, as they are heavily subject to change and usually get affected by the build environment. Hence, features that can be used for direct comparison of binary and source functions usually represent rather abstract properties of these functions.

A small number of abstract features may not seem very usable when a large corpus of candidate source code is considered. However, we have found that the combination of these features produces a sufficiently unique pattern that can be effectively used for either identifying reused functions or narrowing the candidates down to a very small set, which is easy to analyze manually. This key observation has helped us establish a method for direct matching of binary and source functions.

4.3 Function Properties

We leverage a few key properties of functions to form a fingerprint that can be later used to match their source code and binary forms. These properties have one obvious advantage in common: They are almost always preserved during the compilation process regardless of the platform, the compiler, or the optimization level. In other words, despite the feature extraction methods being different for source code and binaries, careful adjustments of these methods can yield the same features being extracted from a function in both forms. These features are as follows:

4.3.1 Function Calls

Functions in a piece of software are not isolated entities and usually rely on one another to for own functionality. A high-level view of the call relations between different functions in a certain program can be represented as a function call graph (FCG), a directed graph in which nodes represent functions and edges represent function calls. In majority of the cases, calls between different binary functions follow the same pattern as in source code. We utilize this fact to combine other seemingly abstract features into sufficiently unique patterns that can be later searched for in source code. There are special cases, most notably inline function expansion (or simply inlining), which sometimes cause this relation not to be easily detectable in binaries. However, as discussed in Section 5.1.3, we employ a technique to minimize the effect, and have actually found inlining not to be a significant limiting factor for our approach in real-world scenarios.

4.3.2 Standard Library and API calls

System calls and standard library function invocations are rather easy to spot in source code. Once function calls are identified, cross-referencing them in each function against the list of declared functions in the same code base separates internal and external calls. Using a list of known system calls and standard library functions,

external calls can be further processed to identify system calls and library function invocations. On the binary side, the situation might be more complex due to different linking techniques. If all libraries are linked dynamically (i.e., runtime linking), the import address table (IAT) of the executable binary yields the targeted system and library calls. While system APIs such as *accept* for socket connections in Linux or those defined in *kernel32.dll* on Windows can be found in the IAT, static linking of the standard library functions such as *memset* results in library function calls appearing like normal internal calls in the binary and not being present in the IAT. However, certain library function identification techniques such as those utilized by IDA Pro's FLIRT subsystem [4] can be used to detect library functions in executable binaries and distinguish them from regular calls between user functions.

4.3.3 Number of Function Arguments

The number of arguments in the function prototype is another feature that can be extracted from source code as well as binaries in a majority of cases. It should be noted however that this is not the case for the actual function prototype as well, since exact type recovery from executable binaries is still an ongoing research problem without fully reliable results [52]. On the source side, parsing the source code easily yields the number of arguments for each defined function. On the binary side, this number can be derived from detailed analysis of the function's stack frame and its input variables combined with identification of the calling conventions used to invoke the function. HexRays Decompiler [6], for instance, uses similar techniques to derive the type and number of arguments for a binary function. While we have found the types not be accurate enough for our purpose, the number of arguments as extracted from binaries is correct in the majority of the cases according to our experiments. An exception to this are functions known as "variadic" functions, such as the well-known "printf" function in C standard library, which can accept a variable number of arguments depending on how they are invoked and the number of passed arguments. However, variadic functions account for a very small fraction of all the functions defined in

real-world scenarios (approximately 1% according to our observations), and they can be treated in a special way to prevent mismatches, as outlined in Section 5.1.5.

4.3.4 Complexity of Control Flow

Control flow can be considered as a high-level representation of a function logic. As discussed in Section 2.1.4, the control flow of each function is represented through a CFG. Predicting the control flow structure of the compiled binary version of a source function without going through the compilation process is extremely unreliable and inaccurate, if not impossible. While the structure of the CFG is usually susceptible to compiler optimizations, we have found that its complexity remains far more consistent between source and binary versions. In other words, simple and complex source functions generally result in respectively simple and complex binary functions, whether or not compiler optimizations are applied.

We use the number of linearly independent paths in a control flow graph to measure its complexity. This metric is referred to as the cyclomatic complexity, and is similarly used in previous work [42] as a comparison metric for binary functions. Cyclomatic complexity is denoted by C and defined as:

$$C = E - N + 2P$$

, where E is the number of edges, N is the number of nodes, and P is the number of connected components of the CFG. In our use case, $P = 1$, as we are measuring the complexity of the control flow structure of individual functions. As can be seen in Figure 5, control structures such as branches and loops contribute to code complexity.

We have carried out an experimental study on the cyclomatic complexity of source and binary versions of approximately 2000 random functions extracted from various projects and compiled with different configurations. The results show that generally, compiler optimizations in fact do not heavily alter the complexity of control flow. Figures 6 and 7 show the correlation between the cyclomatic complexity of source

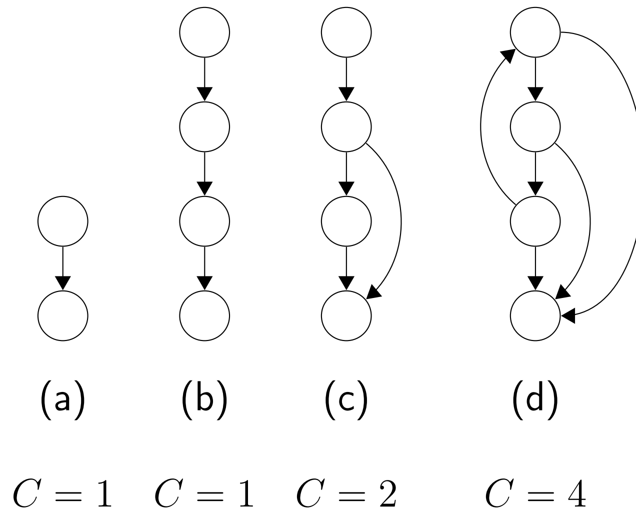

Figure 5: Cyclomatic complexity of four different hypothetical CFGs. c includes a branch at second basic block, and d includes a loop and a break statement at second basic block.

and binary versions, in which each function is denoted by a dot in the scatter graph.

Note how optimizations affect the CFGs of functions by relatively diversifying the graph, but still resulting in a fairly strong correlation between the complexity of source and binary functions. The empty space in the upper left and lower right portions of both graphs, caused by the relative concentration of most dots around the identity ($x = y$) line, shows that cyclomatic complexity can indeed be effectively used as an additional feature for comparing source and binary functions.

For clearer representation of the results, we have only included the functions of which the complexity falls below 150. For 7 functions, accounting for less than 0.4% of all the cases, the complexity of both source and binary control flow graphs is above 150. While they are not depicted in the figures, they follow the same pattern. Among all samples in this study, the maximum difference between source and binary complexities is 32% and belongs to a function with a source CFG complexity of 350.

Our study confirms the fact that predicting the exact cyclomatic complexity for a binary CFG based on the source CFG cyclomatic complexity, or vice versa, is not feasible. However, it also suggests that complexity can still be used as a comparison metric when multiple candidates for a binary function are found. Therefore, in cases

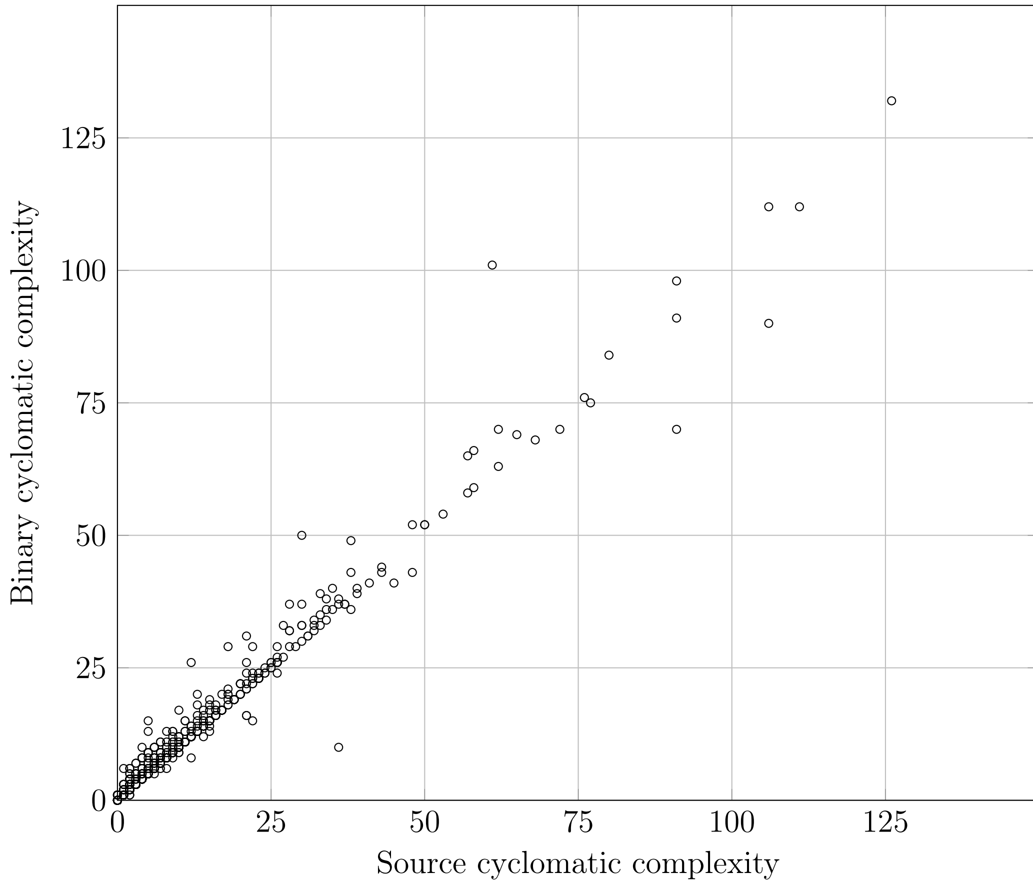

Figure 6: Correlation between the cyclomatic complexity of various functions' control flow in source and binary form, with compiler optimizations disabled.

where multiple candidate source functions are found for one binary function, we will use this metric as a means to reduce the number of false positives and rank the results based on their similarity to the binary functions in terms of control flow complexity.

4.3.5 Strings and Constants

String literals and constants are used in similar work such as RE-Google [50] to match executable binaries with source code repositories. While these two features are certainly usable for such a purpose and can sometimes be used to uniquely identify portions of software projects, we have found them not be reliable enough for function-level matching. For instance, despite the fact that it is relatively easy to extract string literals referenced and used by functions in the source code, we have found

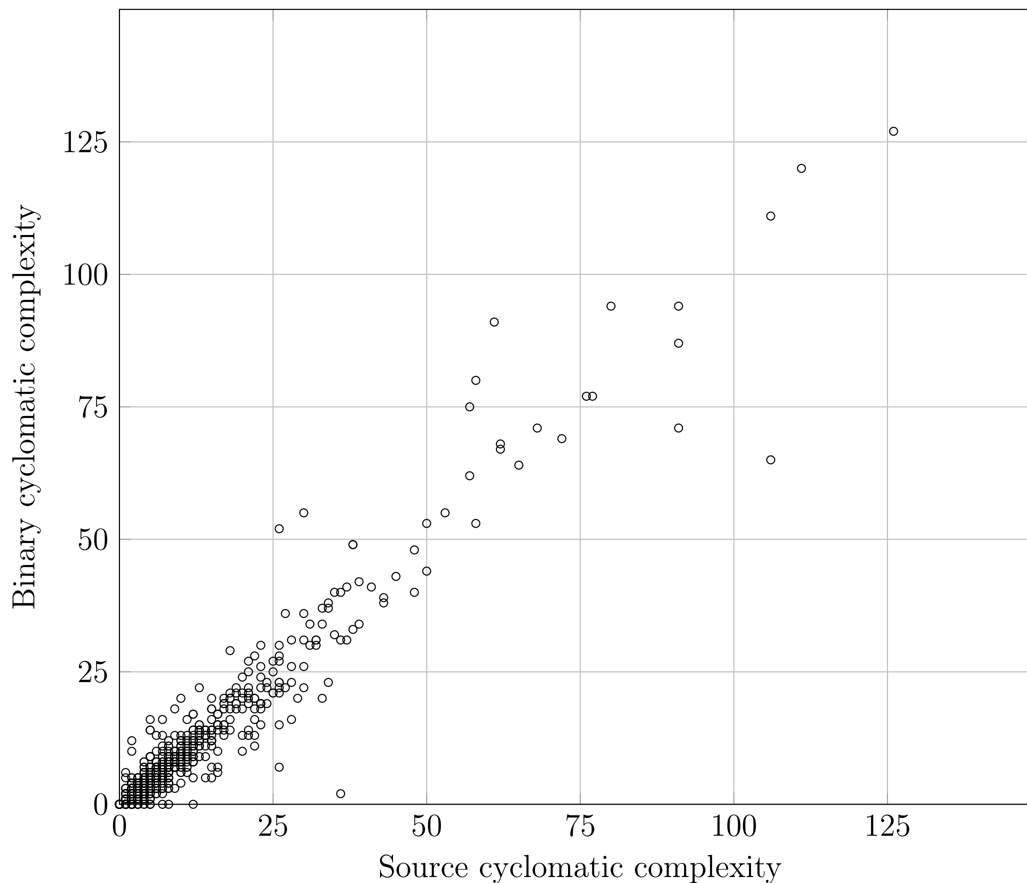

Figure 7: Correlation between the cyclomatic complexity of various functions' control flow in source and binary form, with compiler optimizations set to default level (-O2).

that assigning strings to individual binary functions accurately is rather difficult and does not result in reliable feature extraction. The same issue exists for constants. As a result, we believe string literals and constants can be used in a better way to narrow down the list of candidate projects (and not individual functions), which may actually help in reducing the number of false positives if patterns used by CodeBin exist in more than one software project.

It is notable that the combination of features mentioned above is far more useful than any of them in isolation. For instance, a set of a few API calls may be helpful in narrowing down the list of candidate source functions for a given binary function, but using this technique alone does not lead to many functions being identified, as only

a portion of functions in general include calls to system APIs or standard libraries. On the other hand, while number of arguments is a feature applicable to nearly any function, the set of candidate source functions with a specific number of arguments is still too large to be analyzed manually. However, we show that a carefully designed combination of all these features is sufficient for detecting a large number of binary functions just by parsing and analyzing the source code.

4.4 Annotated Call Graphs

We introduce and utilize the notion of *annotated call graphs (ACG)*, function call graphs in which functions are represented by nodes annotated with function properties. These properties are the features discussed previously, except for the function calls that are represented by the graph structure itself. Hence, an ACG is our model for integrating the features together, forming a view of a piece of source code that is later used to compare it to an executable binary.

In an ACG, functions defined by the programmer, simply referred to as user functions, are represented by nodes. A call from one user function to another user function results in a directed edge from the caller to the callee. Library functions and system APIs may be represented either as nodes or as node properties. If represented by nodes, each called library function or system API will be a node with incoming edges from the user functions that have called it. In this case, calls between library functions may or may not be representable depending on whether the source code for the library is available. If represented by properties, each node (user function) will have a property that lists identifiers for each library function or system API called by that function. As will be discussed later in Section 5.1.2, the decision whether to use nodes or properties for representing library functions and system APIs is critical to the effectiveness of our approach. Temporarily, let's assume than we represent calls to system APIs and library functions as node properties, and a node itself always represents a *user* function.

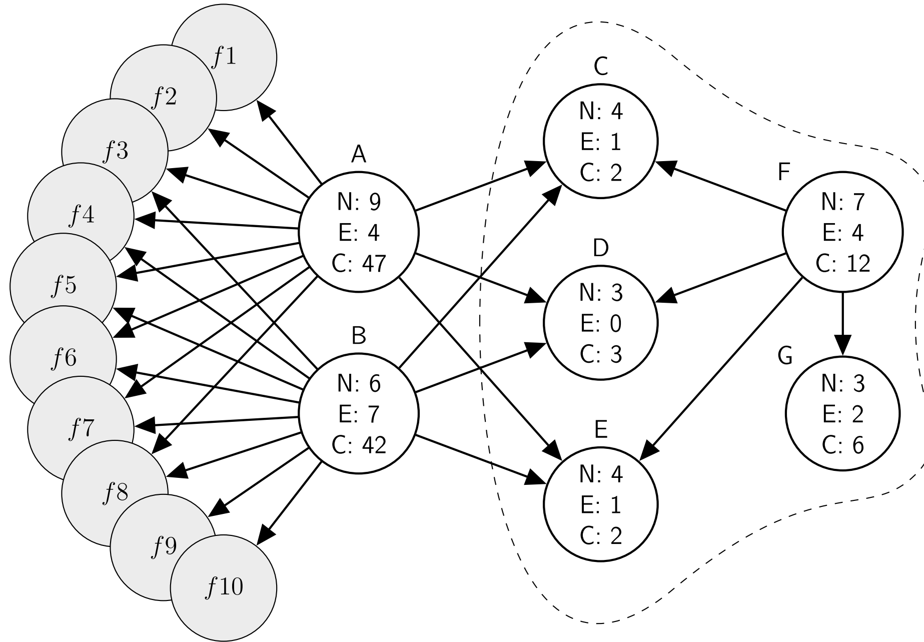

Figure 8: Sample partial annotated call graph from Miniz. N, E and C respectively denote number of arguments, system API and library calls and cyclomatic complexity.

Figure 8 depicts a partial ACG consisting of 17 user functions extracted from Miniz, a relatively simple library that combines optimized compression/decompression and PNG encoding/decoding functionalities, with its complete ACG consisting of 127 nodes and 159 edges. We will use this sample partial ACG to explain our approach advantages and effectiveness. Table 1 includes the actual names of highlighted nodes in Figure 8 for readers' reference.

Each node in the graph is annotated with the features extracted from its respective source function, including the number of function arguments, calls to known standard library functions and system APIs, and the cyclomatic complexity of its control flow graph. Hence, an ACG can be considered a good high-level representation of a software project, combining all its "interesting" characteristics discussed in Section 4.3.

The fundamental idea of our approach for binary to source matching is the following observation: The overall call graph of a piece of software, when augmented by the features discussed before, is fairly unique and generally survives compilation. In

Table 1: Complementary information on Figure 8

Alias	Real Function Name	Library and API calls
A	mz_zip_writer_add_mem_ex	strlen, memset, time, _wassert
B	mz_zip_writer_add_file	memset, strlen, fclose, fread, _wassert, _ftelli64, _fseeki64
C	tdefl_init	memset
D	mz_crc32	
E	tdefl_compress_buffer	_wassert
F	tdefl_write_image_to_png_file _in_memory_ex	malloc, memset, memcpy, free
G	tdefl_output_buffer_putter	realloc, memcpy

many cases, even a small portion of the call graph exhibits unique features.

The outlined portion of the ACG in Figure 8 consists of one function, F, calling four other functions, C, D, E and G. When the number of arguments and the names of recognizable library and system API calls of each of those functions is also taken into account, we have found the pattern to be unique among call graphs of 30 different open source projects, consisting of over 500,000 functions. This basically means that once and if the same pattern is extracted from a binary call graph, it can be searched in many different projects and uniquely and correctly identified.

Although the case highlighted in the paragraph above is a best-case scenario, it is imperative to note that it can be leveraged to identify many more functions as well. For instance, once C, D and E are uniquely identified, such a fact can be effectively used to identify A and B as well. The same idea can be applied once again to identify the functions denoted by $f1$ to $f10$, if they exhibit enough difference in terms of their own properties. Therefore, not all functions need to have very unique and distinctive features or call graph patterns in order to be identifiable. This is in contrast to approaches adopted by previous work on binary to source matching, which can detect reuse only to the extent that unique and identifiable tokens exist in both binaries and source code.

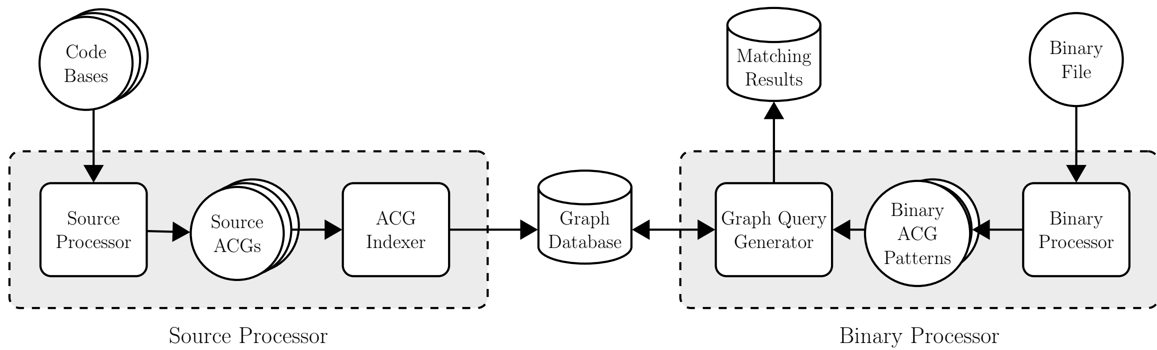

Figure 9: Overall design of CodeBin

4.5 Using ACG Patterns as Search Queries

Based on the observations discussed in this chapter, our approach for matching reused functions in binaries to source code is designed around the following procedure (see Figure 9):

1. **Source Feature Extraction.** First, we parse and analyze the source code and transform it into an ACG by extracting specific features and relationships, such as internal functions calls, number of function arguments, complexity of control flow and calls to standard library functions and system APIs.

2. **Source ACG Storage.** Once an ACG is created from source code, it is stored in a database accompanied by specific indices that allow for fast searching and retrieval. All codebases will be stored in a single graph database with unique labels, allowing for lookup operations over many different codebases.

3. **Binary Feature Extraction.** For each target binary file, we disassemble and analyze the binary to extract the same features as source, creating binary ACG patterns.

4. **Binary ACG Search.** ACG patterns extracted from binary functions are then converted into queries that can be run against the source graph database, effectively searching for similar patterns in all preprocessed source code.

Overall, our adopted method can be compared from various aspects to some previous proposals in the area of binary analysis and software fingerprinting. BinDiff [30, 34, 88] uses similar graph-based techniques to propagate unique features used for matching between two binaries. Joern [85, 84, 86] transforms source code into property graphs and uses graph queries for various analyses on source code. Rendezvous [46] creates a searchable index of binary functions to implement a binary search engine.

Implementing the above procedure bears considerable challenges and complications; and most are unique to the problem of binary to source matching and are caused by the complexities of the compilation process as well as how large projects are usually developed. We discuss these complications and our mitigation techniques in Chapter 5.

Chapter 5

Implementation

In this chapter, we discuss the details of CodeBin, the proof-of-concept implementation of our approach. We also highlight the technical challenges of implementing our approach, and discuss the solutions and workarounds we have integrated into CodeBin to overcome them.

5.1 Challenges

At the core of our approach, we rely on the fact that function call graphs usually follow the same pattern in source code and compiled binaries. There are technical aspects of a compilation process that often change the call graph, effectively making it slightly different than what is perceived from the source code. For the purpose of source to binary matching, it is crucial to either reverse or account for these changes before trying to match call graphs. Below, we discuss these technical aspects and explain how they are mitigated.

5.1.1 Macros and Header Files

As mentioned in Chapter 2.2, we aim at designing a highly automated approach that is ideally capable of handling any code base written in C. This requirement translates into a generic approach for source code processing that is independent of the

build system or other characteristics of the code base. We explained in Section 2.1.1 that defined preprocessor macros and location of header files are essential pieces of information for deriving a correct AST. However, generally, such information cannot be automatically derived from source files. Different projects adopt different layouts for placement of the header files, and many have completely different sets of custom preprocessor macros, all glued together using build scripts that are written for a specific build system.

To better understand the extent of this problem, consider a function defined in a source file that calls another function for which the prototype and implementation are not present in the current source file. If the called function is actually defined in one of the included header files, it is crucial to recognize such a fact, as call graphs form the foundation of our approach. If the header file is not placed in a default directory (e.g., compiler's default directory for header files or the current directory in which the source file resides), a generic approach to source code processing will fail to create the relationship between the caller and the callee. A similar problem exists for "dynamic call expressions", i.e., calls to function pointers that are resolved at runtime and do not represent a concrete call relationship either on the source level or inside the executable binary.

To overcome this problem, we first record all the call expressions encountered inside each function by the name of the called function. The parser also keeps a record of every function declaration (i.e., function prototype) it encounters in all source and header files. When all files are parsed, the list of call expressions for each function is cross-referenced against all declared prototypes, essentially creating valid call graphs and removing dynamic call expressions. A more detailed explanation of these steps can be found in Section 5.2.

An automatic source code processing solution is also likely to miss custom preprocessor macros, which are usually passed along to the compiler using generated build scripts and sometimes are utterly important to derive a correct AST from code. This problem undermines the capabilities of CodeBin in certain circumstances and is

discussed more in Section 7.1.1.

5.1.2 Statically Linked Libraries

As discussed in Section 4.3, we enumerate calls to certain standard library functions to form one of the features of each function. Standard libraries, just like any other library, have a chance of being statically linked to the target program during the linking process. During static linking, the linker treats libraries as part of the code, effectively inserting library functions into the binary. This differs from dynamic (runtime) linking, in which only the symbol names are inserted into the final executable.

Static linking results in significant changes in the call graph, which is demonstrated in Figure 10. Calls to statically linked library functions change the call graph by creating additional nodes and corresponding edges for each library function. As a result, not only the binary call graph is different from the source code call graph, but also the library calls cannot be easily extracted from binary functions just by enumerating the symbols used as call targets.

There are three possible solutions to the problem of statically linked libraries, all of which are based on the idea of normalizing the call graph patterns extracted from source code, binary, or both.

1. The first solution is to change the source call graph, effectively inserting library functions as separate nodes instead of properties of their callers. In other words, the first solution is to process the source code with the assumption that standard libraries will always be linked statically. Even when the libraries are actually linked in a dynamic fashion, the binary call graph can be easily adjusted to represent static linking, simply because the names of library functions can be retrieved by enumerating imported symbols. The obvious advantage of this approach is that it requires no additional information other than a list of standard library functions, and it bears low technical overhead. On the other hand, this approach shifts the matching criteria more towards the call graph pattern and

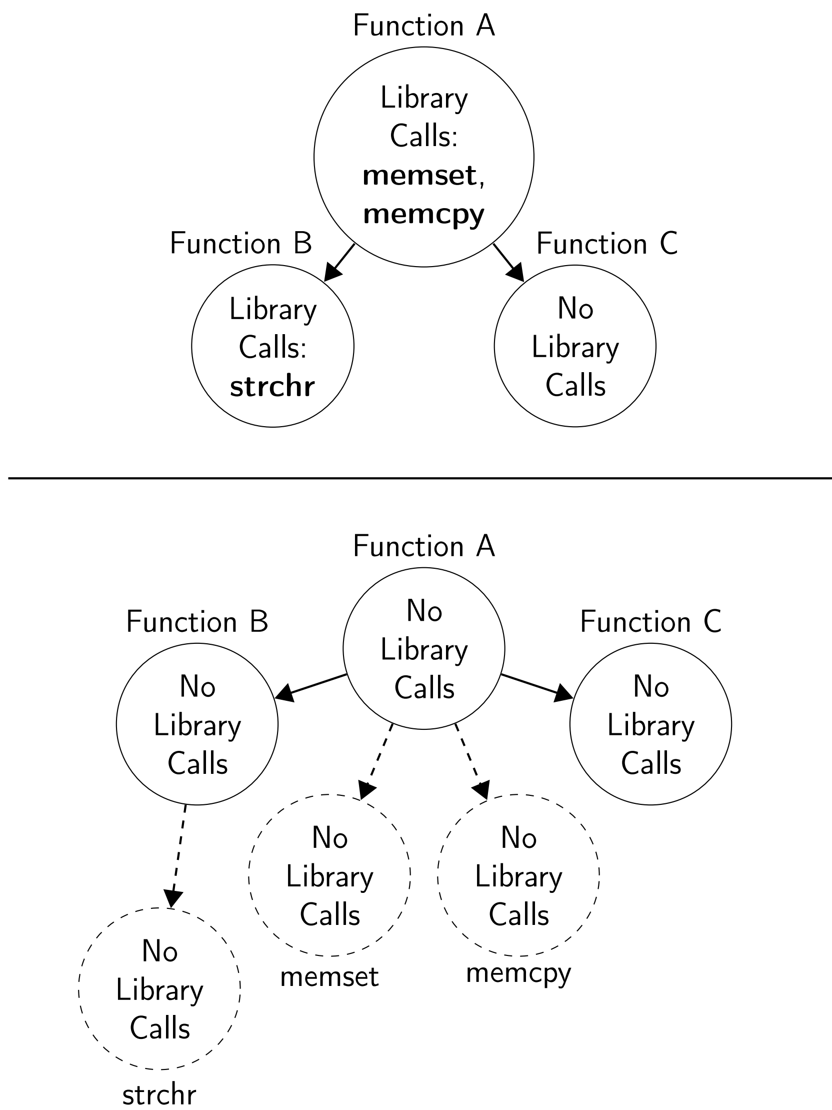

Figure 10: The effect of static linking on binary ACGs. A, B and C are user functions. Top: Dynamically linked libraries. Bottom: Statically linked libraries.

away from the features of individual nodes, leaving the nodes (functions) with even less identifiable features.

2. Another solution is utilizing tools such as FLIRT to identify library functions in binaries, and adjusting the binary call graph accordingly. In this approach, the source processor always inserts library functions calls as node properties into the call graph. The advantage of this solution compared to the first one is that individual functions keep one of their important properties, making them more

identifiable when an entire pattern is found. The downside is that this solution relies heavily on the library function identification technology to be robust and reliable, which is not always true. For instance, while FLIRT identifies some specific library functions rather reliably, it may fail in identifying others. Such a failure at the presence of a statically linked library results in the wrong graph pattern being extracted, which in turn causes a mismatch.

3. The third solution combines the first two and aims at bringing the best of both into one solution. With this technique, the source processor is made aware of the capabilities of the library function identification technology used. This can be done through a simple configurable list of library functions, which includes all the functions that can be reliably detected using the mentioned technology. If a library function call is seen on the source side, the source processor either inserts it as a node property or a separate node, depending on whether the function is included in the list or not, respectively. With this approach the capabilities of existing technologies for binary clone detection are used for handling statically linked libraries, while their occasional failures do not negatively affect the results of binary to source matching. CodeBin adopts this approach to overcome the problem of statically linked libraries.

5.1.3 Function Inlining

Inline function expansion or simply inlining is a form of compiler optimizations that replaces a call to a function with the function body, eliminating the call and making the callee part of the caller function. While there are ways to instruct compilers to enforce or avoid function inlining [3], it is often up to the compiler, which in turn depends on the chosen optimization level among other factors. Since inlining has complicated effects on performance [25], prediction of it happening for a specific function is rather difficult. Generally, a small function that is not called by many other functions has a better chance of being inlined.

Function inlining changes the ACG extracted from binaries by eliminating the connection representing the call to the inlined function, and adding the inlined function's library and system API calls into the callers' features.

To understand the effect of inlining on our approach, it is important to note that ACG search queries are generated from binaries and performed over source call graphs. These queries can be broken down into two integral components: Graph structure (nodes and edges) and node properties. Search queries will match the structure of the source ACG if the structure of the source ACG remains a superset of the binary ACG structure, which is still true when inlining happens. However, library and system API calls will create a mismatch. Such a situation can be partially avoided by making the lookup operation for this feature work in reverse: Names of library functions and system APIs called by a source function that is matched in terms of the ACG structure will be looked upon in the names of APIs called by the binary function, but not vice versa. As a result, if the binary function includes calls to other libraries and system APIs as a result of inlining one of its callees, a mismatch is avoided.

However, two issues still remain: (i) Mismatch of the feature representing standard library function calls when a library function gets inlined, and (ii) change to the cyclomatic complexity of the binary function. The latter will only be an issue if another candidate with the same features and similar cyclomatic complexity is found for the binary function. While we have not been able to remedy these issues completely, we have found both of them to be of minor effect in the overall accuracy of CodeBin.

5.1.4 Thunk Functions

Thunk functions, also known as jump functions, are helper functions generated and inserted into assemblies by compilers. Thunk functions typically contain very few or simply a single jump instruction, and are used for a variety of reasons. For instance,

Microsoft Visual Studio C/C++ compiler with the "incremental linking" option enabled (which is so by default), inserts jump thunks into binaries to minimize the time needed to relocate a function during runtime after load. Other usages of thunk functions include conversion of calling conventions, or implementing virtual calls in object oriented languages such as C++.

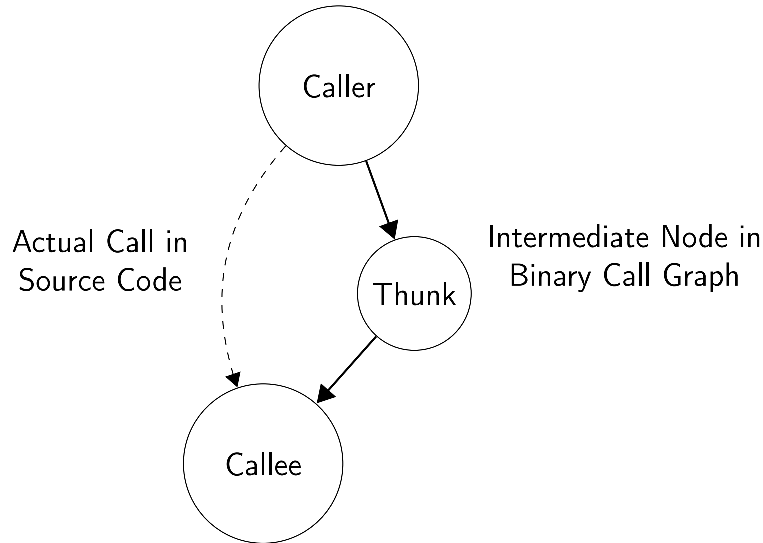

Figure 11: Thunk functions

Nevertheless, thunk functions interfere with call graph matching as they introduce intermediate nodes into the binary call graph (see Figure 11). During feature extraction from binaries, the probable presence of such functions should be taken into account, and intermediate nodes should be recognized and removed before searching for the extracted binary ACG. CodeBin adjusts the binary call graph in the presence of thunk functions by identifying and removing them. Identification of thunk functions is done by relying on IDA Pro function flags as well as a simple analysis of function instructions.

5.1.5 Variadic Functions

Variadic functions are defined to accept an undetermined number of arguments. A very well-known example of such a function is `printf`, which writes a series of bytes

into the standard output by receiving a template and a variable number of arguments based on the output template. The actual number of arguments for a variadic function as observed in a binary file is indeterministic and depends heavily on how the function is called. As CodeBin uses the number of arguments as an exact feature of call graph nodes, variadic functions should be handled in a special way so that such a fact does not result in mismatches.

CodeBin implementation includes such special handling by detecting variadic functions during parsing and incorporating a special boolean property for each node in the derived ACG. This property is dedicated to variadic functions and is set to true only when the node represents such a function. A small tweak in graph query generation ensures that the number of arguments is only matched if the variadic property of the node is set to false, i.e., the function is not variadic.

5.2 Source Code Processing

5.2.1 Preprocessing and Parsing

Automatic generation of the annotated call graphs from source code requires a robust preprocessor and parser. We utilize Clang, a full fledged open source C/C++ front-end for the LLVM compiler infrastructure, to parse and process source codes. Clang exposes some of its functionalities through a high-level API, which can be used for general high-level processing of C/C++ and Objective C code. However, more detailed functionalities such as creation of control flow graphs from syntax trees are not available via the high-level API. The LLVM framework is written in a fully modular fashion, and one can use any part of its internal functionalities for miscellaneous purposes, commonly referred to as “tooling”. CodeBin uses both techniques to leverage Clang functionalities in parsing and reasoning about source code.

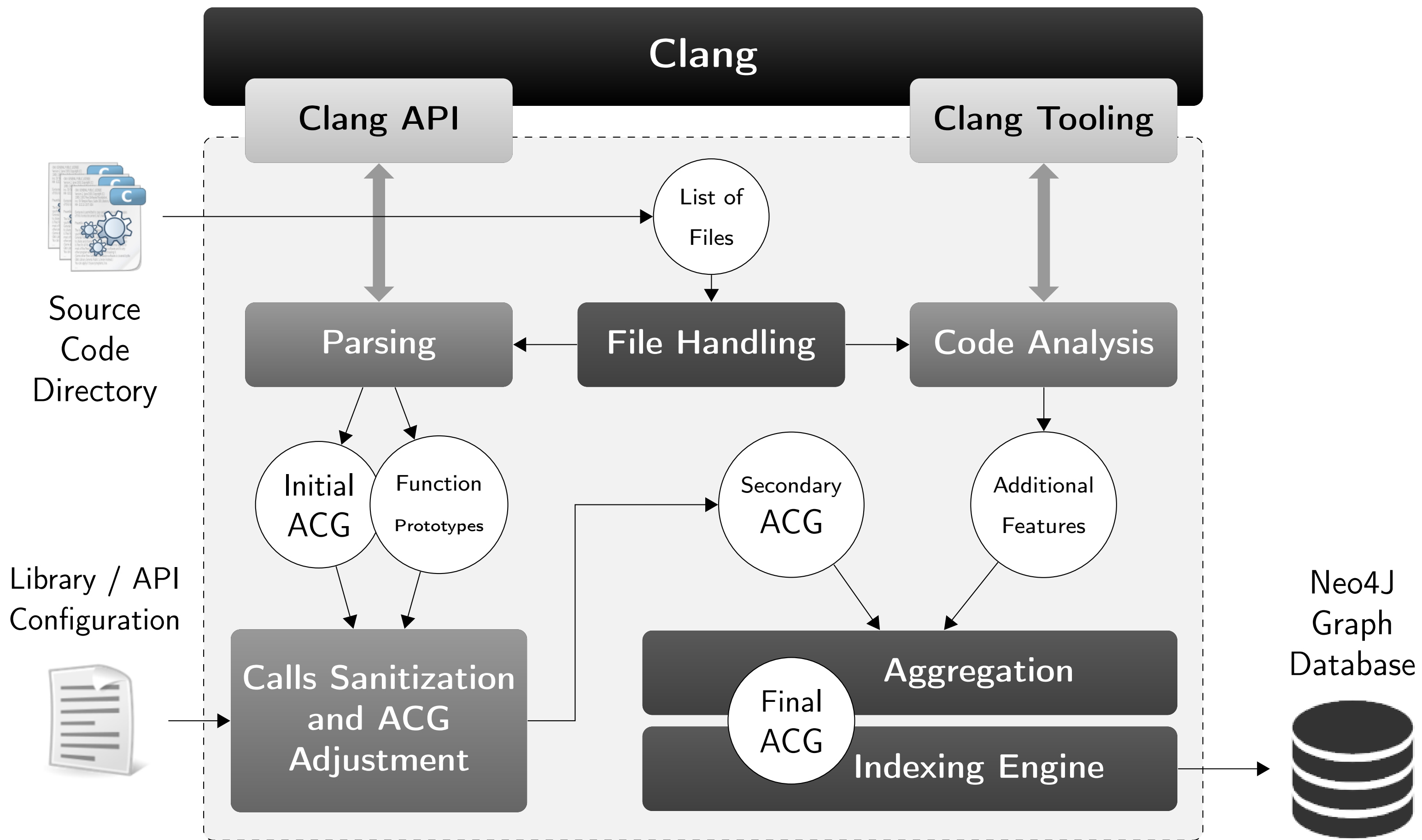

Figure 12: The architecture of CodeBin source code processor.

5.2.2 Source Processor Architecture

The architecture of CodeBin source processor is depicted in Figure 12. It is mostly developed in Python 2.7, except for the "code analysis" module that is a Clang tool written in C++. CodeBin source processor can be used in a completely standalone mode, with no need for Clang or the LLVM framework being present on the system. CodeBin processes code bases one at a time. User inputs to the source processor are the path to the code base, a name for the project being indexed, and any additional compilation switches (such as predefined macros).

A file handling module iterates over all the files in the specified path, targeting C source code and header files. A list of found source files is then separately fed to parsing and analysis modules. The parsing module invokes the Clang parser on each file, extracting function definitions, prototypes and call expressions using pre-registered callback functions. The parsing module distinguishes between library functions and user functions by the path of the file in which the function definition is found, and only parses user functions. Up to this point, a set of individual functions is created, each having several properties such as a name, an exact location, as well as the number of arguments. Once all function definitions are visited, call expressions of each function are enumerated in order to separate internal and external calls. Any function call with a "known" target (i.e., one that is already defined amongst user functions) is treated as an internal call, creating directed edges between the nodes representing the caller and the callee. All other call expressions are temporarily treated as being external. This set includes calls to library functions as well as dynamic call expressions. Dynamic call expressions are calls of which the target will be determined during runtime. An example of such an expression is a call performed via a function pointer. The output of the parsing module is an initial annotated call graph as well as a list of all the function prototypes visited, whether they are defined by the user or exist in a library.

On the other side, the code analysis module uses a similar technique for parsing source files and identifying user functions, this time using Clang internal libraries

instead of its high-level API. This module leverages the Clang library to generate source control flow graphs from the abstract syntax tree of each function, and calculates their cyclomatic complexities. This modules also can determine whether a function accepts a variable number of arguments (i.e., a variadic function), thanks to Clang library capabilities that are only available through tooling. The output of this module is a set of two features, namely cyclomatic complexity and whether the function is variadic, for each user function. These additional features are then integrated with other features during the next steps.

The outputs of the parsing module, as well as a set of configuration files for library functions and system APIs are used by the call sanitization and call graph adjustment module to generate a refined annotated call graph. This module performs two main tasks:

1. It identifies and removes external calls that are a result of dynamic call expressions based on the list of all visited function prototypes it has received as the second input.

2. It adjusts the call graph and normalizes the library calls based on the configuration files provided for library functions and system APIs. This step, as discussed in Section 5.1.2, is crucial to have usable call graphs.

The call graph adjustment module outputs a modified, secondary ACG. The function properties extracted by the code analysis module now have to be integrated with this ACG, a task that is performed by the aggregation module. This module creates a finalized version of the ACG, which includes all the function properties and is adjusted to deal with special scenarios such as statically linked libraries.

Finally, an indexing module is responsible for storing the final ACG into a graph database. This module transforms CodeBin's internal representation of the ACG into a format that is understood by the backing database, and stores it so that it can be queried later by the binary analysis module.

5.3 Binary File Processing

Decoding and disassembly of binary files is a complicated process. We use IDA Pro as a platform for binary analysis, utilizing and building upon many of its capabilities using its scripting engine, IDAPython [8]. We rely on IDA pro for disassembling binary files, recovering functions and control flow graphs, recognizing calling conventions and stack frame analysis, identifying cross-references in the binaries, parsing import address tables, and identification of common standard library functions using its FLIRT subsystem.

Our binary file processing engine is written as a Python plugin for IDA Pro, and introduces additional analysis passes on a binary file assembly instructions. These analysis passes enable the CodeBin plugin for IDA Pro to form a high-level view of the disassembled binary, including binary function ACGs.

5.3.1 Extracting Number of Arguments

CodeBin incorporates two different components for extracting the number of arguments for binary functions. One of these components performs a stack frame analysis partially by using IDA Pro capabalities. The other component invokes the HexRays Decompiler plugin for IDA Pro and counts the number of arguments on the function prototype reported by the decompiler. During our experiments, we have found both methods to be fairly accurate for 32-bit x86 binaries. However, the latter component considerably outperforms the former on 64-bit binaries in terms of accuracy, while being also significantly slower, especially for larger functions. CodeBin includes a configuration option that allows the user to choose the second component for extracting the number of function arguments, provided that the decompiler plugin is installed.

5.3.2 ACG Pattern Extraction

CodeBin matches binary functions to source functions by searching for partial ACG patterns. ACG patterns may be extracted in several ways. However, CodeBin uses

a very specific method: For each binary function, a partial ACG pattern is created by extracting features from the function and all the other functions called by it. In other words, each ACG pattern represents a function and its immediate callees, all annotated by their respective features. The rationale for adopting this method is based on the following observations:

1. CodeBin is designed to detect code reuse, where only portions of a specific program might be reused. Since a function relies on all the other functions it calls to perform its operation, callees of a function are very unlikely be changed when it is reused in another program for performing the same operation. On the other hand, callers of a function may be changed based on the context the function is used for. As a result, the ACG pattern extracted from a function and its callers may not be found in a the previously indexed source code containing the target function, even when it is actually copied. Based on this observation, CodeBin forms an ACG for each function only based on its callees.

2. During our experiments, we have found that our implementation does not always yield the correct features of binary functions. This limitation is discussed more in Section 7.1.3. Inaccurate feature extraction from a single function in an ACG pattern results in the whole pattern not being matched to source code ACG, effectively lowering the chance of correct matching for all the other functions in the pattern. While enumerating callees of the callees of a function, i.e., going more than one level deep in the call graph, theoretically yields a more unique pattern; we have found it to have non-negligible negative effect on querying performance as well as matching accuracy due to occasional inaccurate feature extraction. As a result, CodeBin only enumerates the immediate callees of each function, and adopts a different method discussed in Section 5.4.1 for leveraging the uniqueness of bigger call graph patterns.

Similar to fine-grained adjustments performed on source ACGs (see Section 5.2), CodeBin may additionally modify the extracted ACGs to bring them closer to source

ACGs. For instance, if a call to a statically linked library function identified by IDA FLIRT is detected, CodeBin removes the node representing the library function and adds its name to the list of library functions called by the caller function, which forms one of its features and is treated as a node property. Other minor adjustments on node properties such as normalizing the names of library functions is also performed.

Once an ACG is extracted from a binary function and its immediate callees, it is converted into a query that can be run against the source graph database.

5.4 Graph Database

Currently, we use Neo4J that is a general purpose graph database to index the annotated call graphs we extract from source code. We have chosen Neo4J mainly due to its querying capabilities with the Cypher query language. As discussed in Section 5.1, we introduce slight complications in the queries we run against the graph data store to compensate for possible differences between the binary and source call graphs, e.g. those caused by function inlining.

At the core of our approach, we are relying on lookup operations on call graphs. In these search operations, we are representing the subjects (ACG patterns) by nodes relationships and properties. Therefore, it is necessary for our back-end data store to be able to perform fast lookups based on graph features, and to expose an interface for representing such partial graph patterns. Neo4J is therefore an ideal solution, as it indexes the graphs by the features we use in our queries and also supports a query language tailored to graph pattern lookups.

5.4.1 Subgraph Search

Cypher is a declarative query language for Neo4J and allows for complex lookups to be performed over Neo4J graphs using relatively simple queries. Partial ACGs extracted from binaries are first transformed into Cypher queries by the CodeBin IDA Pro plugin and then executed against the source ACG indexed by Neo4J. Cypher queries

are generated in a way to represent CodeBin's searching approach, including tweaks introduced for handling variadic functions and possible changes in node properties due to inlining. These queries also contain carefully designed aggregations to allow concise representation of the results, in case a binary ACG pattern matches many different source ACG patterns and returns a large number of results.

```
MATCH a-->b,a-->c,a-->d,a-->e

    WHERE a.nargs = 7 OR a.variadic=TRUE
    AND ALL (ea IN ["free","malloc","memset","memcpy"]
    WHERE ea IN a.excalls)

    AND b.nargs = 3 OR b.variadic=TRUE
    AND ALL (eb IN ["realloc","memcpy"]
    WHERE eb IN b.excalls)

    AND c.nargs = 4 OR c.variadic=TRUE
    AND ALL (ec IN ["_wassert"]
    WHERE ec IN c.excalls)

    AND d.nargs = 4 OR d.variadic=TRUE
    AND ALL (ed IN ["memset"]
    WHERE ed IN d.excalls)

    AND e.nargs = 3 OR e.variadic=TRUE

RETURN collect(distinct id(a)
                 + "|" + labels(a)[1]
                 + "|" + a.name
                 + "|" + a.file
                 + "|" + a.line
                 + "|" + a.column
                 + "|" + a.complexity) as a,
       collect ...
```

Listing 3: Cypher query for the ACG pattern in Figure 8

Listing 3 includes part of the Cypher query generated from the outlined pattern in Figure 8. Part of the `RETURN` expression is left out for brevity.

As can be seen in Listing 3, binary functions are represented by aliases in generated Cypher queries. CodeBin keeps track of a mapping between binary functions,

represented by their addresses in the binary, to aliases in each query and uses it as a reference to assign candidate source functions to binary functions once it receives and parses the results of running queries.

5.4.2 Query Results Analysis

It is likely for binary functions to appear on more than one ACG pattern. For instance, in Figure 8, C, D and E will be part of three patterns, generated from F, A and B. As a result, more than one set of candidate source functions may be assigned to one binary function. In these cases, CodeBin analyzes and compares the candidate sets two by two. If two candidate sets have a non-empty intersection, only the candidates in their intersection are kept and the rest are ruled out. If the intersection is empty, it simply means that both candidate sets can be equally correct, and CodeBin will keep all candidates in both sets until the binary function appears in yet another pattern.

Finally, when all the patterns are generated and their respective queries are run, CodeBin will rank the candidates for each binary function according to their similarity to the binary function in terms of control flow complexity. CodeBin uses an adaptive threshold to remove the candidates with cyclomatic complexities that are highly unlikely to be correct, but only if more probable source candidates are already found for the binary function.

5.5 User Interface

Except for the source code processing and indexing engine, the rest of CodeBin is implemented as an IDA Pro plugin. This plugin exposes the following functionalities of CodeBin through a user interface integrated into IDA Pro:

1. **Source code parsing and indexing.** CodeBin plugin for IDA Pro allows users to invoke the source processing engine on a code base by pointing to its root directory. When a name is chosen for the code base and optional preprocessing

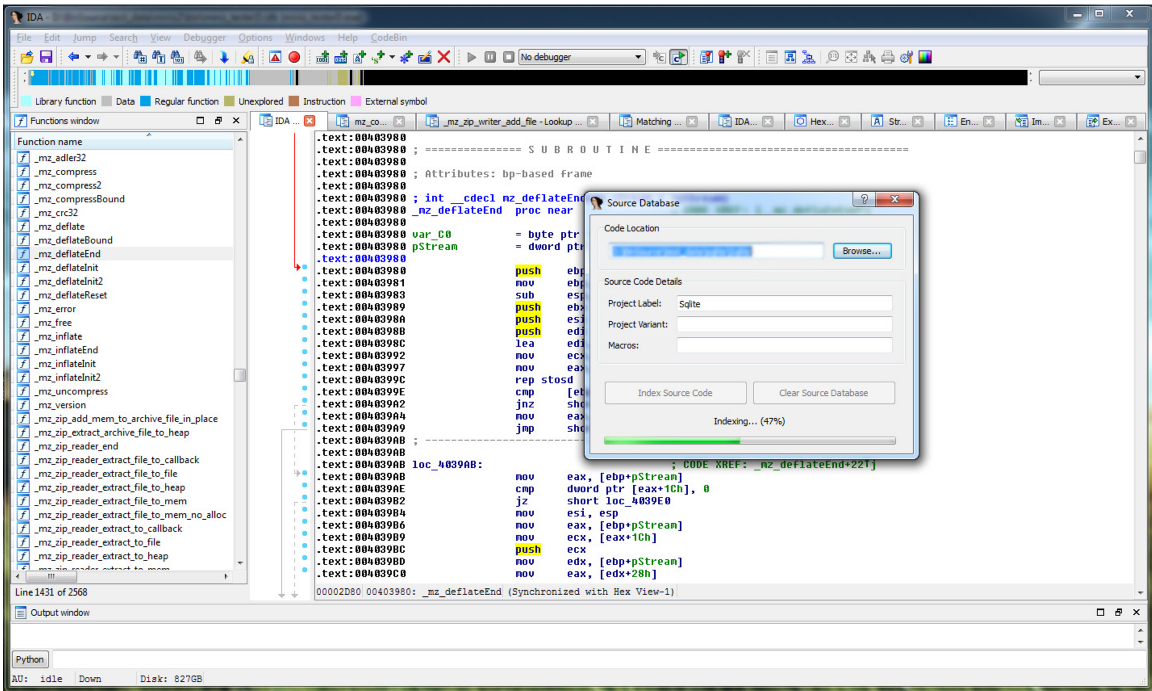

Figure 13: User interface: Indexing source code.

macros are given, the code base will be automatically parsed and indexed into the graph database. This part of the interface is depicted in Figure 13.

2. **Viewing partial ACGs.** As shown in Figure 14, it is possible for users to conveniently inspect partial ACGs extracted from arbitrary binary functions using extra commands inserted into the context menu of IDA Pro *Functions View* widget. Users can also copy the respective Cypher query for each partial ACG, as well as a simplified version of it. The simplified version is not optimized, but is easier to understand and run in Neo4J interactive web console.

3. **Selective matching of binary functions to source code.** Similar to viewing partial ACGs, the plugin also allows users to selectively match individual binary functions to source functions. By invoking this command, ACG patterns will be extracted from selected binary functions and then converted into Cypher queries, which are then run against the source graph database. The results will be processed to match returning source candidates to binary functions, and can

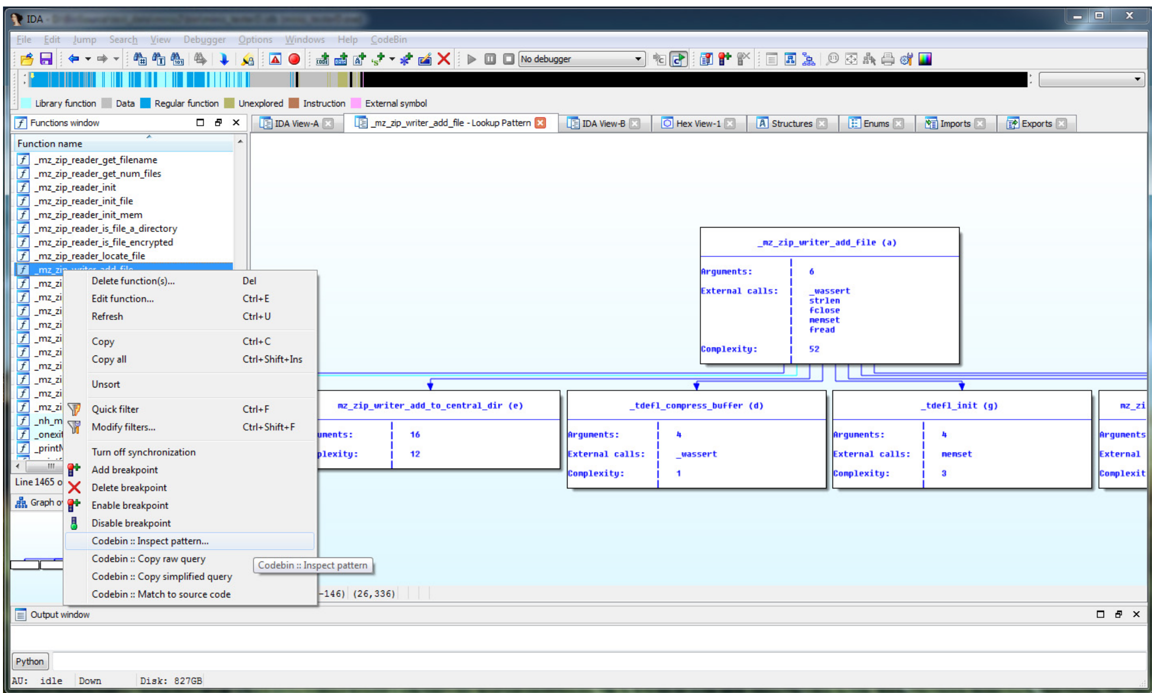

Figure 14: User interface: Inspecting ACGs.

later be viewed using the next feature.

4. **Inspection of results.** The plugin lays out the results of binary to source matching in a table, allowing users to see all the returning source candidates for each binary function, as well as their similarity to the binary function in terms of cyclomatic complexity. An example is depicted in Figure 15.

5. **Viewing source code.** CodeBin plugin for IDA Pro also allows the users to see the source code for each matching candidate without leaving IDA Pro. As shown in Figure 16, the source code is syntax-highlighted and represents the code exactly as written in the code base, with all the original comments and before any preprocessing is performed.

Similar to modern IDA Pro plugins such as DIE [18], CodeBin utilizes PySide [65] shipped with IDA Pro 6.8 and later to provide user interface widgets integrated with IDA Pro. Call graph patterns are displayed using IDA Pro built-in *Graph View*, and source code syntax highlighting is performed using Pygments [21].

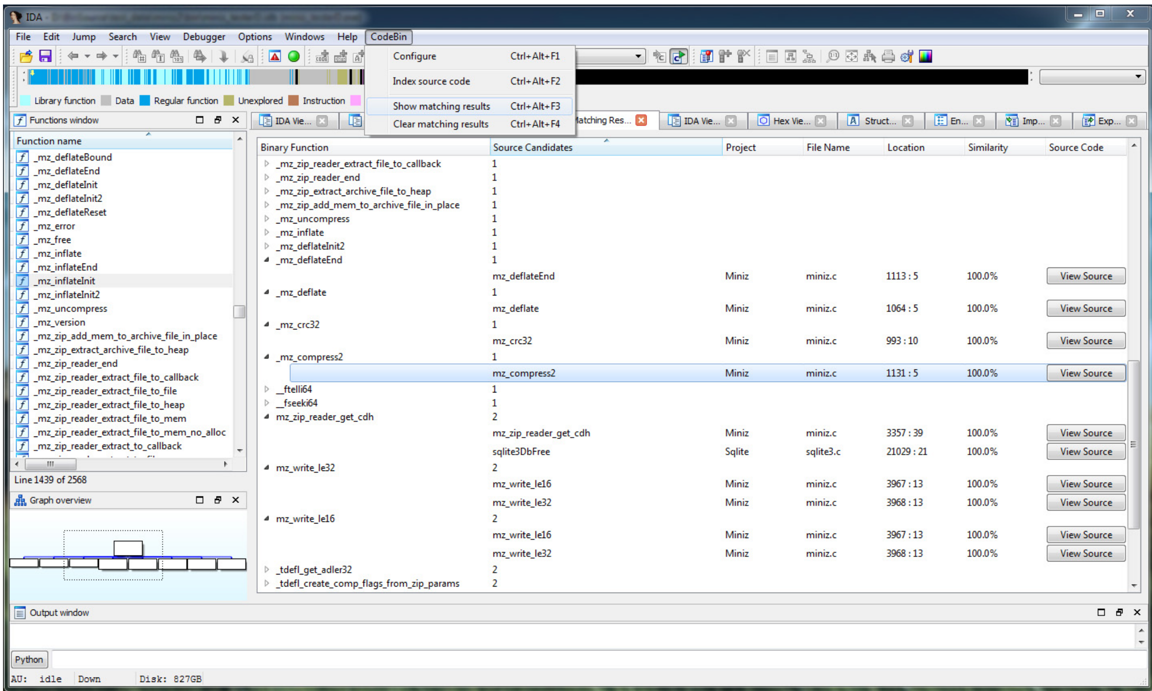

Figure 15: User interface: Viewing matching results.

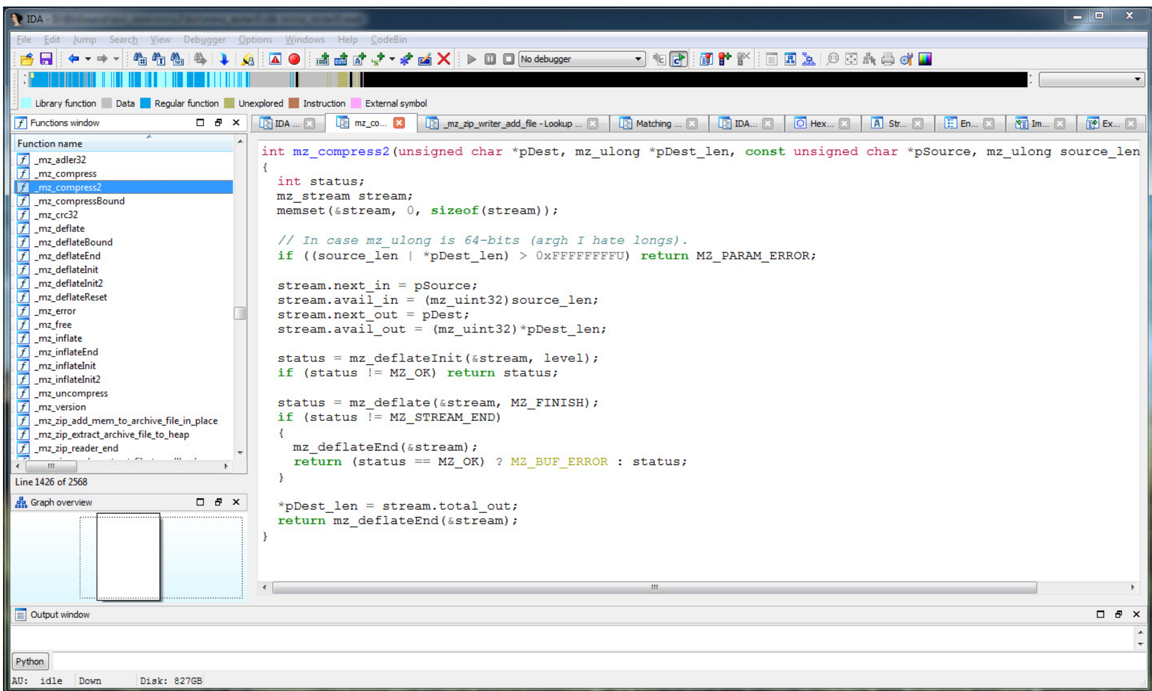

Figure 16: User interface: Viewing source code.

Chapter 6

Evaluation

We have evaluated CodeBin in a scenario that is similar to real-world reverse engineering settings. In this scenario, the whole or parts of specific projects for which the source code is available may be reused in a program for which only the executable binary is available. The goal is to leverage CodeBin for detecting code reuse and automating the binary to source matching process as much as possible. By evaluating CodeBin in such a scenario, we explore the following questions:

1. Can CodeBin be used to match binary functions to source functions in real-world programs compiled with realistic settings?

2. Are the features used in CodeBin enough to uniquely identify or significantly narrow down reused functions amongst tens of different projects and several millions lines of code?

3. How does CodeBin perform when there is no reuse?

4. Is CodeBin able to perform similarly on binaries compiled with different compilers and/or for different operating systems?

5. How viable is automatic binary to source matching for reverse engineering, and what are the challenges?

Overall, our experiments show that CodeBin has the potential to identify a significant number of reused binary functions in source code. Localizing searches through the call graph by using previously identified functions is shown to be fairly effective. In the rest of this chapter, we present the methodology, test cases and data, and the results of our evaluation.

6.1 Methodology

6.1.1 Test Scenario

To evaluate CodeBin, we have downloaded and indexed a total of 31 open source projects, accounting for 524,168 source functions and 24,344,652 lines of code. These projects were obtained from several sources such as Github [5] and SourceForge [14]. No specific decision factor is taken into account for deciding on which projects to choose among open source software. That being said, we have tried to incorporate all kinds of projects including standalone applications, special-purpose solutions, libraries and system tools to test CodeBin in a general setting.

We have used 11 executable binaries, 7 of which reuse all or portions of previously indexed projects. The other 4 binaries do not reuse any of the indexed source code. To be able to establish this ground truth, we have chosen binary versions of programs for which the code is open and available, and have manually confirmed that they do not include any part of indexed source code. From the first 7 binaries, we have compiled 2 programs on two different operating systems (Linux and Windows) using two different compilers (GCC 4.9.2 and Microsoft Visual C++ 12.0), to see how CodeBin is affected by changes in these settings. All tests are run on a Windows 7 x64 desktop machine powered by an Intel Core i7-4790 CPU with a frequency of 3.6GHz utilizing 8GB of memory and regular SSD storage.

Currently, we only target C source code and x86 binaries; however, CodeBin does not rely on any functionality of IDA Pro that is specific to x86 binaries, and therefore adopting it to other binary file formats and platforms should be straightforward. We

do not however limit the build process by choosing specific optimization levels or other compilation options, and the default settings are used for all binaries. Our results show that except for slight differences in compiler's decision to inline certain functions that in turn slightly affects the extent of identified functions, CodeBin's overall performance is not susceptible to changes in such compilation options.

6.1.2 Pattern Filtering

Since we evaluate our approach in a scenario when hundreds of thousands of source functions have an equal chance of being reused in a target binary, searching for very simple call graph patterns results in tens or hundreds of results being returned. Manually inspecting such a high number of results one by one is arguably a very tedious and error-prone process. In some cases, manual analysis of a binary function to understand its operations may even be quicker. On the other hand, searching for these patterns by running queries and analyzing the large result sets returned also considerably slows down CodeBin.

As a result, we have implemented a feature in CodeBin to filter out simple call graph patterns. This is a custom feature and it is possible to query all ACG patterns by default. CodeBin assigns a score to each extracted ACG based on the presence of distinctive features, such as library and API calls, high number of function arguments and the total number of binary functions included in the pattern. A threshold is then defined and used to filter out simpler patterns, so that only distinctive ones are converted into queries and searched for. This feature also filters out the patterns generated from standard library or compiler functions, in part by relying on IDA FLIRT [4] flags.

Hence, the total number of patterns extracted and searched for from each binary is smaller than the total number of binary functions in an executable. This feature significantly speeds up the search process and helps to achieve more useful results.

6.1.3 Result Collection and Verification

The evaluation is performed as follows: First, we have processed and indexed the source code of all 31 projects in the graph database (see Section 6.5). Then, for each binary file in our collection of 10 candidate binaries, CodeBin has extracted ACG patterns from all binary functions and run the filtering process. Once relatively distinctive patterns are identified, CodeBin has searched for them by converting them into and executing Cypher graph queries. Eventually, result sets are analyzed as discussed in Section 5.4.2 and reported.

We have used debugging information as the ground truth to evaluate and measure the accuracy of CodeBin in identifying reused parts of code (i.e., functions). CodeBin, however, does not use any debugging info for extracting features from binary functions, and equally works on stripped binaries. It is important not to use debugging information, since most real-world executable binaries are stripped from any such information. By running CodeBin on stripped binaries and establishing ground truth by manually comparing 50 binary functions in 3 projects compiled with and without debugging information, we have confirmed the similarity of the results on stripped and non-stripped binaries.

Finally, using the ground truth established based on debugging information, we have reviewed and verified the matching results as reported in the next section.

6.2 Evaluation Results

Results of evaluating CodeBin on binaries that reuse parts of previously indexed source code is listed in Table 2. For each binary file, we have included the total number of functions that are also present in our source base. Due to inlining of simpler functions, partial reuse of a project's code in some binaries, and other compilation complexities such as dead code elimination, this number is different from the total number of source functions in the main reused project. Total number of candidate ACG patterns for searching is also included for each binary, which may include some

Table 2: Results of evaluating CodeBin in real-world scenarios.

Reused Project	Total Reused Functions	Queried ACG Patterns	Unique and Correct	Correct Candidate in Top 3	Correct Candidate in Top 5	Correct Candidate in Top 10	Too Many Candidates (20+)	Mismatched / False Positive	Not Matched	Total Running Time (min:sec)
Miniz	114	131	65.8%	70.2%	76.3%	78.9%	1.8%	2.6%	16.7%	1:14
Sqlite	1391	682	74.5%	78.6%	81.3%	84.4%	0.6%	1.2%	13.8%	6:41
Silver Searcher	66	95	68.2%	75.8%	77.3%	81.8%	1.5%	4.5%	12.1%	1:07
Redis	2329	1532	65.2%	75.1%	80.2%	86.8%	0.5%	2.1%	10.6%	11:51
Coreutils	1856	1194	53.4%	64.9%	73.4%	76.4%	1.9%	2.2%	19.5%	9:08
PCRE	342	74	31.2%	41.8%	49.4%	55.8%	1.8%	3.5%	38.9%	3:55
OpenSSL	3982	2163	9.4%	11.3%	12.6%	14.7%	0.6%	1.4%	83.3%	23:42
OpenSSL (manual)	3982	2163	62.0%	64.9%	68.1%	72.7%	6.8%	1.2%	25.5%	27:16

statically linked standard library functions missed by our filtering process.

Once all queries are executed and results are analyzed, we have verified the results, which is a list of mappings between binary functions and one or more source candidates, using debugging information (function names and parent project). In the results list, a binary function may be correctly matched to a unique source candidate (*Unique and Correct* results). It may be mapped to several source candidates, in which the correct source function is in the top 3, 5, or 10 candidates. As expected, for some binary functions the ACG patterns have not been distinctive enough, resulting in the list of candidates having more than 20 source functions. Some reused functions have not been matched to any source candidate, either because the pattern has not been distinctive enough and was filtered out, or because the search query has not returned any results. In some cases, binary functions are matched to one or more source candidates, all of which have been wrong, which account for *Mismatched / False Positives*.

For OpenSSL, we have conducted the experiment twice. In the first run, we have processed the source code in a fully automated setting, without specifying any preprocessor macros. In the second run, we have removed all the OpenSSL nodes in the graph database, and reprocessed the source code with carefully chosen preprocessor macros so that they exactly match the ones specified during the compilation process. While the second setting results in significantly better results, choosing the correct set of preprocessor macros cannot be performed automatically and requires manual work. For further discussion on this issue, see Section 7.1.1.

Our evaluation results show CodeBin's potential for speeding up the reverse engineering process. For instance, note that for a program that uses Sqlite, over two thirds of reused functions (1036 out of 1391) are uniquely and correctly matched to their source code. This process has taken less than 7 minutes to complete on a moderately powerful workstation and the original source code, potentially with descriptive comments, are shown to the reverse engineer afterwards. Less than 2% of all the returned results are false positives in this case. Similar results have been observed

Table 3: CodeBin results in no-reuse cases.

Project	Binary Functions	Matched ACG Patterns	Matched Functions
Pidgin	2618	63	8.7%
Custom Program	1153	16	4.9%
FFMpeg Utility	974	28	8.6%
HT Editor	527	19	8.2%

for Miniz, Redis, Silver Searcher and Coreutils. In some cases, CodeBin has also identified several statically linked library functions missed by FLIRT, thanks to the technique discussed in Section 5.1.2.

In the rest of this section, we report and investigate CodeBin's results on binaries that do not include any reused functions, as well as binaries compiled in different environments.

6.3 No Reuse

4 out of 11 executables used in our evaluation do not reuse any part of code from the 31 indexed projects, a fact that is known due to manual inspection of their source code. Our custom program is created by mixing 92 random functions and statically linking the C standard library to the compiled binary. Clearly, CodeBin may still return matches in these cases, as there is always a chance of multiple ACG patterns in different projects, previously processed or not, to possess the same structure and features. This situation can be considered similar to when portions of previously indexed source code is actually reused in a binary, but the source code is then taken out of the source database before running CodeBin on the binary. In this case, one or more source candidates will be returned for some binary functions, except for the fact that the correct candidate will not be in the results list, as it is now absent in the source database.

Table 3 includes the results of running CodeBin on each of the 4 binaries that do not reuse any part of previously processed source code. *Matched Patterns* denotes the total number of queried patterns that have returned results, and *Matched Functions* shows the total number of binary functions for which at least one candidate is returned in the results list.

These results show a noticeable difference compared to cases where part of the code is actually reused, as the total number of matched functions compared to the number of binary functions is significantly lower. However, for a fairly large binary like Pidgin, the number of matched functions (all of which are false positives in this case) is still high considering that no reuse is taken place. Mitigating this issue in general requires identifying and leveraging more features from binary and source functions, which may also help in reducing the number of candidates returned for a binary function when it is actually reused.

This may cause a problem compared to cases where a small number of binary functions are reused, making it difficult to distinguish between small reuse and false positives in no-reuse. A relatively noticeable difference however is seen in the size of matched ACGs. False positives in no-reuse cases usually form small, isolated ACGs scattered through the binary. When some functions are actually reused, they usually form bigger coherent ACGs representing the piece of functionality that is reused.

6.4 Different Compilation Settings

To test the effect of different compilation settings on CodeBin's performance, we have compiled *Miniz* and *Sqlite* for both Windows and Linux using Microsoft Visual C++ Compiler (MSVC) 10.0 (2008) and 12.0 (2013) and GCC 4.7 (released in 2012) and 4.9.2 (released in 2015) under three different optimization levels, and then run CodeBin on all binaries and compared the results (see Table 4). Overall, we have made the following observations:

- **Different optimization levels.** Disabling compiler optimizations consistently

Table 4: Effect of different compilation settings on CodeBin's performance.

Project	Compiler Used	Optimization Level	Total Reused Functions	Total Candidate Patterns	Unique and Correct	Mismatched / False Positive	Not Matched	Total Running Time
Miniz	MSVC 12.0	O2	108	130	69.4%	2.8%	18.5%	1:04
		Ox	101	128	65.3%	4.9%	16.8%	1:03
	GCC 4.9.2	O2	114	131	65.8%	2.6%	16.7%	1:08
		O3	99	125	63.4%	4.0%	14.1%	1:04
Sqlite	MSVC 12.0	O2	1391	682	74.5%	1.2%	13.8%	6:31
		Ox	1342	664	69.2%	2.5%	12.6%	6:18
	GCC 4.9.2	O2	1427	693	73.7%	1.4%	12.5%	6:49
		O3	1359	671	64.9%	2.1%	11.8%	6:24

improves CodeBin's performance and results in accuracy figures higher than what is reported in Tables 2 and 4. However, we do not consider these cases realistic, as the reason for disabling or selecting a low optimization level is usually ease of debugging during the development phase [12]. Optimizing for size (*Os* flag on both GCC and MSVC) slightly improves the results compared to level 2 optimization (*O2* flag on both compilers), as it prevents the compiler from inlining some of the functions in favor of the total size of the binary. On the other hand, performing full optimization in favor of speed (*O3* and *Ox* flags on GCC and MSVC) results in very aggressive inlining decisions and impacts CodeBin accuracy. We have included the results for level 2 optimization (usually the chosen level in projects build settings) and full optimization for speed (the

most challenging case for CodeBin) in Table 4.

- **Different compiler versions.** We have found changing the compiler version to have the least effect on CodeBin's performance. Binaries compiled with the same level of optimization but using different versions of the same compiler are distinctively different in terms of the overall layout, binary instructions and introduced compiler functions. However, call graphs and ACGs extracted by CodeBin from these binaries are usually the same and the differences in matching results are negligible.

- **Different compilers/platforms.** the total number of reused binary functions slightly varies as the compiler changes, due to different inlining decisions. We have observed more aggressive inlining performed by MSVC compared to GCC, which has in turn slightly reduced the number of recognizable functions by CodeBin because of less distinctive patterns. Nevertheless, CodeBin performs very similar on both versions, resulting in close numbers in terms of both matched and mismatched reused functions. These results are expected, as almost all of the differences caused by different binary formats and other variations are abstracted away by IDA Pro. CodeBin uses the same interface provided by IDAPython to analyze and extract features from binaries, and is therefore automatically able to operate in the same way over different executable files as long as IDA Pro retains its disassembly and analysis capabilities.

6.5 Source Base and Indexing Performance

We have collected a total of 31 open source C projects to form a source base for our evaluation experiments, which are then processed and indexed into a Neo4J graph database by CodeBin source processor. This source base consists of 24,344,652 lines of C code as counted by cloc [28], excluding blank lines and comments. The graph

database consists of 548,023 nodes, with 524,168 nodes representing functions identified within the source code of the indexed projects by CodeBin. Other nodes, as explained in section 5.1.2, represent standard library functions.

Our source base is deliberately made very large by including several big projects (3 OS kernels including Linux kernel and large projects such as Wireshark, OpenSSL, Tor, Git, etc) to test CodeBin's ability in searching on big datasets. We have measured the time required by CodeBin to preprocess, parse, analyze and index each project source code. On average, CodeBin source processor has spent barely over a second for completely processing each 1000 lines of code (again, excluding blank lines and comments). It should be noted that due to online updating and creation of indexes by Neo4J, the time required for indexing source code expectedly increases as the dataset grows larger.

Table 5 includes the number of functions, number of lines of code, and indexing time for each of the indexed projects.

Table 5: CodeBin test dataset, parsing and indexing performance.

Project	Functions	Lines of Code	Indexing Time
Linux Kernel	236,083	13,416,043	2:29:14
Wireshark	67,216	2,917,350	0:51:38
GCC Compiler	52,531	2,480,461	1:04:53
Minix 3	89,057	2,305,133	1:08:21
Inferno OS	28,694	672,553	0:19:42
Hashkill	2,782	420,719	0:02:38
Vim	9,481	311,865	0:11:09
OpenSSL	4,697	280,826	0:03:37
Apache HTTPD	3,496	188,807	0:02:11
Tor	4,241	179,282	0:05:27
Git	5,880	157,858	0:07:11
Tengine	706	165,620	0:00:32
Nginx	547	119,517	0:00:26
MPV Player	4,105	107,411	0:03:25
Sqlite	1,859	99,674	0:01:31
Unqlite	2,682	84,347	0:02:05
Putty	2,275	80,882	0:01:53
PCRE	394	71,241	0:00:21
Coreutils	2,124	59,673	0:00:56
STB	452	56,899	0:00:21
Redis	2,952	46,407	0:01:49
MozJPEG	1,058	44,028	0:00:48
JQ	571	17,624	0:00:27
Zmap	360	11,638	0:00:19
Memcached	296	10,931	0:00:12
LMDB	199	9,837	0:00:12
Curl	64	9,434	0:01:03
Miniz	120	5,260	0:00:07
Wrk	138	4,481	0:00:07
RobotJS	98	4,438	0:00:23
Silver Searcher	75	4,413	0:00:13
Total	524,168	24,344,652	6:42:12
Average	16,908	785,311	0:12:58

Chapter 7

Discussion

In this section, we investigate the results of evaluating CodeBin and highlight its limitations through actual examples. We also discuss CodeBin from a security perspective, and provide possible directions for future work towards mitigating its limitations.

7.1 Limitations

7.1.1 Custom Preprocessor Macros

OpenSSL heavily relies on custom preprocessor macros to customize implementations on different environments and optimize performance. This technique includes snippets of code written to operate on a set of parameters just like regular functions, but instead defined using macros. Many functionalities in OpenSSL have multiple implementations, some defined using parameterized macros and others as regular functions. Values for these macros are then defined during the build process to incorporate one of the implementations for each functionality into the binary, depending on the compiler, platform, and operating system.

Listing 4 shows four different implementations for a bit rotation function in

OpenSSL, defined using a parameterized macro specified as *ROTATE*. These different definitions along with similar others result in more than 10 different versions of the *des_encrypt1* function in OpenSSL with 4 different call graph patterns. Each of these versions is optimized for a specific environment and will be put into the executable according to macros passed to the compiler.

```
# if (defined(OPENSSL_SYS_WIN32) && defined(_MSC_VER))
#     define ROTATE(a,n)       (_lrotr(a,n))

# elif defined(__ICC)
#     define ROTATE(a,n)       (_rotr(a,n))

# elif defined(__GNUC__) && __GNUC__>=2 &&                   \
       !defined(__STRICT_ANSI__) && ...

#     if defined(__i386) || defined(__i386__) ||             \
#       defined(__x86_64) || defined(__x86_64__)

#           define ROTATE(a,n) ({ register unsigned int ret; \
                                  asm ("rorl␣\%1,\%0"        \
                                     : "=r"(ret)             \
                                     : "I"(n),"0"(a)         \
                                     : "cc");                \
                                  ret;                       \
                               })
#     endif

# endif

# ifndef ROTATE
#     define ROTATE(a,n)       (((a)>>(n))+((a)<<(32-(n))))
# endif
```

Listing 4: Different implementations for ROTATE in OpenSSL.

Extended use of this technique, as observed in OpenSSL, significantly undermines CodeBin's capabilities in a fully automated setting. If the same set of preprocessor macros are not used for compiling the binary program and parsing the code using CodeBin's source processor, different implementations mentioned above almost always

result in significant differences in the call graph structure as well as the function properties, hence causing mismatches during partial ACG searches. When the same set of preprocessor macros are used, the results significantly improve, as can be seen in Table 2.

7.1.2 Orphan Functions

Another challenging case for identification of reused functions using CodeBin's approach is caused by dynamically invoked functions. In C, function pointers are utilized to perform such dynamic invocation. The target of a call operation that is performed via a function pointer is only determined during runtime. Hence, no assumption can be made about the target of such call during parsing or even compilation. While sophisticated inter-procedural data flow analysis techniques might help in identifying the targets of dynamic calls during compilation, dynamic analysis is usually considered the only robust solution in similar cases [31].

Functions that are only invoked via function pointers therefore result in "orphan" nodes in the source call graph, i.e., nodes that are isolated and not connected to the rest of the graph. While they exhibit the same behavior in a binary call graph extracted via static analysis, an isolated function effectively disallows using call graph relations to augment its feature set. These orphan functions are extremely unlikely to match to anything less than hundreds of candidates due to their very small single-node ACG, regardless of their own feature set.

PCRE is a library for incorporating Perl-compatible regular expressions into C programs, and heavily relies on dynamic function invocation. Based on the call graph generated by CodeBin, 184 out of 394 functions in its source code are orphan. As a result, CodeBin has failed to detect over 33% of reused PCRE functions. We consider such a case to be an inherent limitation of static analysis rather than binary to source matching, and argue that a reliable solution for handling such cases is highly unlikely to be achievable without symbolic execution or dynamic analysis.

7.1.3 Inaccurate Feature Extraction

In a relatively small number of cases, mismatches or false positives are simply caused by inaccurate feature extraction, i.e., the binary ACGs having incorrect properties. A majority of these cases can be further broken down into two main categories:

1. **Incorrect identification of number of arguments.** CodeBin uses the output of the HexRays decompiler to determine the number of arguments for each binary function, with a fallback mechanism that incorporates a heuristic based on stack frame analysis and is invoked if the decompiler plugin is absent. Tests are done on 32-bit binaries using the fallback mechanism, which has the advantage of speed. However, it results in an incorrect number of arguments being extracted for some binary functions. While the accuracy of these two mechanisms are comparable on 32-bit binaries, HexRays decompiler relatively retains its accuracy while the fallback mechanism fails in many cases on 64-bit binaries.

2. **Incorrect demangling of symbol names.** CodeBin incorporates a mechanism to demangle and normalize the names of invoked system APIs and standard library functions. This mechanism fails to extract the correct name in some cases, resulting in incorrect properties in binary ACGs.

Due to the fact that these problems have limited negative effects compared to custom preprocessor macros and orphan functions, we have not investigated them further in our current implementation.

7.1.4 Similar Source Candidates

The results show that in many situations, CodeBin returns multiple source candidate functions for a binary function. While CodeBin ranks the candidates for each binary function based on their similarity to the binary function in terms of control flow complexity, the correct candidate does not end up in the top 3 or top 5 in some cases. By manual inspection of some of such results, we have found that CodeBin

```c
int SQLITE_STDCALL sqlite3_column_count(sqlite3_stmt *pStmt) {
    Vdbe *pVm = (Vdbe *)pStmt;
    return pVm ? pVm->nResColumn : 0;
}
```

```c
static int strlen30(const char *z) {
    const char *z2 = z;
    while( *z2 ) {
        z2++;
    }
    return 0x3fffffff & (int)(z2 - z);
}
```

Listing 5: Similar functions in Sqlite

usually runs out of features to further distinguish between possible candidates in these cases. A simple example for such a case in *Sqlite* is shown in Listing 5, where two functions, *sqlite3_column_count* and *strlen30*, both called from *do_meta_command*, share exactly the same features: Both have the same number of arguments of 1, the same control flow complexity of 2, and neither call any other function.

To human eyes, these source functions still look distinguishably different: They employ different control constructs (a loop versus a condition), they have different types for the arguments, and the latter includes a distinctive constant (`0x3fffffff`) while the former includes a very common one (zero). However, annotating functions in a large database with features that differentiate them in terms of such properties bear significant challenges. See Sections 2.1.5, 4.3.3 and 4.3.5 respectively for further discussion on these issues.

7.1.5 C++ Support

CodeBin is currently only capable of extracting ACGs from C code, and does not yield correct results for C++ code. We highlight three major challenges of adopting our approach to C++ code.

1. **Late/Dynamic Bindings.** Dynamic bindings of function call targets in C++ results in an issue similar to dynamic function invocations in C, while being much more common due to techniques like polymorphism and virtual functions. Handling these cases on source code requires dependency analysis, which is unlikely to be possible without compilation. On binaries, virtual table call resolution code precedes the actual call instruction and should be analyzed and reasoned about, an analysis that is outside the scope of this work.

2. **Library Function Identification.** We have found FLIRT performance on C standard library functions to be relatively good with approximately 80% of statically linked functions being identified on average. CodeBin uses these results to annotate binary ACGs and create more distinctive patterns. However, according to our experiments, FLIRT shows very low accuracy on C++ standard library functions, with over 90% of linked functions (606 out of 668) missed in a sample executable binary. C++ templates further intensify this problem, as they result in compile-time code generation that is very likely to impose significant challenges on a signature-based detection system such as FLIRT.

3. **Overloaded Functions.** On the source side, CodeBin relies on function names to resolve internal calls and distinguish external calls across whole codebases without necessarily having access to the location of header files. Since function names are unique descriptors in C, this approach results in fairly accurate call graphs while having the advantage of being fully automatic. In C++, function names are not enough to resolve a call target and much more entities such as prototypes, namespaces, classes and object references should be taken into account. Such an analysis, as performed by compilers, requires data flow analysis and is not possible to perform on potentially non-compilable code.

7.2 CodeBin as a Security Tool

Binary analysis and reverse engineering tools are in part used for security purposes, such vulnerability detection and tracking, exploit generation, malware analysis and clustering, etc.

However, many such tools, including CodeBin, are not primarily designed as a security tool. The reason is twofold: First, popular obfuscation techniques like binary packing are usually dealt with before any actual binary analysis is done [37, 79]; for example by processing the obfuscated binary through a toolchain that unpacks the binary and reverses known obfuscation techniques. Second, many automated analyses do not succeed at completely addressing unintentional complicating variables such as different computing platforms and build environments [31]. In these cases, an unknown obfuscation technique may simply render the problem inexplicable in an automated way, at least until robust techniques are developed for handling non-malicious cases.

This basically means that these tools are not developed with a strong threat model in mind, and do not try to be robust against intentional evasion. In this particular case, for example, CodeBin is not designed or evaluated against an adversary who actively tries to conceal code reuse. For instance, adding dummy arguments in random function prototypes and performing dummy operations on these variables would be a relatively easy way to change ACGs and make then unrecognizable by CodeBin, even though the main functionality of the reused code stays the same.

7.3 Directions for Future Work

CodeBin establishes a base for binary to source matching, augmenting syntactic tokens and text-based searches. According to what is perceived from its evaluation, CodeBin can be extended in several directions, all of which may well introduce new challenges.

Detecting build systems and leveraging configuration and make scripts will probably bring a key improvement to CodeBin. We argue that for binary to source matching to retain its advantages, processing of source code should remain fully automatic. Our approach currently incorporates several techniques to work around the limitations caused by this requirement, but does so by compromising accuracy in certain cases as discussed in Section 7.1.1. In the presence of an automatic system for detecting popular build systems such as GNU make, CMake or MSBuild, a large number of open source projects can be parsed more accurately by leveraging the exact compiler invocation commands along with a tool like CIL.

CodeBin can also be improved by leveraging symbolic execution and dynamic analysis techniques in certain cases, as highlighted in Section 7.1.4. While these techniques are not without their disadvantages in terms of execution speed, they may still be beneficial if they are proven to help in extracting additional features from binary functions such as referenced strings and constants.

Chapter 8

Conclusion

Reverse engineering is a demanding and complex process, and requires manual effort in many cases. It is shown in previous work that certain aspects of binary analysis and fingerprinting such as clone and reuse detection can be automated to aid the process and make it more efficient.

We explored a relatively new and less explored idea in the area of clone-based reverse engineering based on automatic binary to source matching. We characterized the challenges by elaborating on the important aspects of the software build process that affect the similarity of an executable binary program to its source code. Through several experiments, we also identified certain features that can be reliably extracted from both source code and binary code and used for comparison in an automated fashion.

Based on our findings, we designed a graph-based approach to binary to source matching that is based on relative uniqueness of program call graphs augmented by high-level function properties. We discussed the challenges and technical complications of implementing this approach, and introduced several workarounds and mitigation techniques so that the process can remain fully automatic. Through designing realistic evaluation scenarios, we showed that our approach is capable of detecting a significant number of reused functions in executable binaries by only processing the source code while requiring relatively low computing resources and time. Eventually,

we elaborated on the limitations and provided directions for future work towards further improvements.

Based on the results of our experiments, we argue that binary to source matching can be considered as a complementary approach to clone-based reverse engineering, and has the potential of revealing code reuse while not being affected by many issues that undermine binary clone detection techniques. We hope that our work establishes a robust base for future research in this direction.

Bibliography

[1] Boomerang: A general, open source, retargetable decompiler of machine code programs. SourceForge project. http://boomerang.sourceforge.net.

[2] CMake. Build system for C/C++ projects. https://cmake.org.

[3] Common function attributes. GNU Compiler Collection online documentation. https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html.

[4] Fast Library Identification and Recognition Technology (F.L.I.R.T.). Online article (May 2015). https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.

[5] GitHub. https://github.com.

[6] Hex-Rays Decompiler: Overview. Online article. https://www.hex-rays.com/products/decompiler/index.shtml.

[7] IDA: About. Online article. https://www.hex-rays.com/products/ida/.

[8] IDAPython project for HexRay's IDA Pro. GitHub project. https://github.com/idapython/src.

[9] Node package manager (npm). Project website. https://www.npmjs.com.

[10] Ohloh API documentation. Online article (2014). https://github.com/blackducksoftware/ohloh_api.

[11] Open Hub. https://www.openhub.net.

[12] Options that control optimization. GNU Compiler Collection online documentation. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[13] RE-Google aids code analysis. H-Online Security Blog (Nov. 2009). http://www.h-online.com/security/news/item/RE-Google-aids-code-analysis-862539.html.

[14] SourceForge. http://sourceforge.net.

[15] The Decompilation Wiki. Online article. http://www.program-transformation.org/Transform/DeCompilation.

[16] F. E. Allen. Control flow analysis. In *Symposium on Compiler Optimization*, New York, NY, USA, 1970.

[17] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *European Conference on Software Maintenance and Reengineering (CSMR'07)*, Amsterdam, Netherlands, Mar. 2007.

[18] Y. Balmas. Dynamic IDA enrichment. GitHub project. https://github.com/ynvb/DIE.

[19] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance (ICSM'98)*, Bethesda, MD, USA, Nov. 1998.

[20] M. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, Nov. 1994.

[21] G. Brandl, T. Hatch, and A. Ronacher. Pygments. Project website. http://pygments.org.

[22] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification*, Snowbird, UT, USA, July 2011.

[23] D. Cabezas and B. Mooij. Detecting source code re-use through a binary analysis hybrid approach. Online article (Feb. 2013). http://www.forensicmag.com/articles/2013/02/detecting-source-code-re-use-through-binary-analysis-hybrid-approach.

[24] A. Caliskan-Islam, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. Technical report (Dec. 2015). https://www.princeton.edu/~aylinc/papers/caliskan-islam_when.pdf.

[25] W. Y. Chen, P. P. Chang, T. M. Conte, and W.-m. W. Hwu. The effect of code expanding optimizations on instruction cache designar. *IEEE Transactions on Computers*, 42(9):1045–1057, Sep. 1993.

[26] C. Cifuentes. Structuring decompiled graphs. Technical report (1994).

[27] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, University of Queensland, 1994.

[28] A. Danial. CLOC - count lines of code. GitHub project (2016). https://github.com/AlDanial/cloc.

[29] E. Dolgova and A. Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105–119, Mar. 2009.

[30] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC'05)*, Rennes, France, June 2005.

[31] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security Symposium*, San Diego, CA, USA, Aug. 2014.

[32] M. R. Farhadi, B. Fung, P. Charland, and M. Debbabi. BinClone: Detecting code clones in malware. In *International Conference on Software Security and Reliability (SERE'14)*, San Francisco, CA, USA, June 2014.

[33] H. Flake. The StrucRec plugin. IDA-Pro Plugin. https://www.hex-rays.com/products/ida/support/freefiles/strucrec.zip.

[34] H. Flake. Structural comparison of executable objects. In *International Workshop on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'04)*, Dortmund, Germany, July 2004.

[35] P. S. Foundation. Pypi - The Python Package Index. https://pypi.python.org/pypi.

[36] I. Guilfanov. Decompilers and beyond. In *BlackHat Security Conference*, Las Vegas, NV, USA, Aug. 2008.

[37] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in Computer Virology*, 6(3):261–276, Aug. 2010.

[38] A. Hemel. Binary analysis tool. Online article. http://www.binaryanalysis.org/en/home.

[39] A. Hemel. Introducing the binary analysis tool. Online slide set. http://events.linuxfoundation.org/sites/events/files/slides/bat.pdf.

[40] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, USA, Feb. 2014.

[41] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *ACM Conference on Computer and Communications Security (CCS'11)*, Chicago, IL, USA, Oct. 2011.

[42] J. Jang, M. Woo, and D. Brumley. Towards automatic software lineage inference. In *USENIX Security Symposium*, Washington, DC, USA, Aug. 2013.

[43] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *Program Protection and Reverse Engineering Workshop (PPREW'14)*, San Diego, CA, USA, Jan. 2014.

[44] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Working Conference on Reverse Engineering (WCRE'08)*, Antwerp, Belgium, Oct. 2008.

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

[46] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Working Conference on Mining Software Repositories (MSR'13)*, Piscataway, NJ, USA, May 2013.

[47] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Symposium on Static Analysis (SAS'01)*. Springer, Paris, France, July 2001.

[48] C. Lattner. LLVM and clang: Next generation compiler technology. In *The BSD Conference*, Ottawa, ON, Canada, May 2008. `http://clang.llvm.org`.

[49] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization (CGO'04)*, Palo Alto, CA, USA, Mar. 2004. `http://llvm.org`.

[50] F. Leder. RE-Google. Online article (2009). `http://regoogle.carnivore.it`.

[51] F. Leder. RE-Google plugin documentation. Readme file. `https://www.hex-rays.com/contests/2009/REGoogle/README.TXT`.

[52] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, USA, Feb. 2011.

[53] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Network and Distributed System Security Symposium (NDSS'12)*, San Diego, CA, USA, Mar. 2010.

[54] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[55] C. Lindig. Random testing of c calling conventions. In *International Symposium on Automated Analysis-driven Debugging (AADEBUG'05)*, Monterey, CA, USA, Sep. 2005.

[56] K. Lu, D. Muller-Gritschneder, and U. Schlichtmann. Hierarchical control flow matching for source-level simulation of embedded software. In *International Symposium on System on Chip (SoC'12)*, Tampere, Finland, Oct. 2012.

[57] V. Massol and T. M. O'Brien. *Maven: A Developer's Notebook: A Developer's Notebook*. O'Reilly Media, Inc., 2005. `https://maven.apache.org`.

[58] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *GCC Developers' Summit*, Ottawa, ON, Canada, May 2003.

[59] J. J. Miller. Graph database applications and concepts with Neo4J. In *Southern Association for Information Systems Conference (SAIS'13)*, Atlanta, GA, USA, Mar. 2013.

[60] A. Mockus. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07*, Washington, DC, USA, May. 2007.

[61] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil - infrastructure for c program analysis and transformation (v. 1.3.7). `https://www.cs.berkeley.edu/~necula/cil/`.

[62] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CC'02)*, Grenoble, France, Apr. 2002.

[63] K. Noyes. Open source is driving business app development, survey finds. PCWorld Online Article. `http://www.pcworld.com/article/254296/open_source_is_driving_business_app_development_survey_finds.html`.

[64] A. Prakashm, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Network and Distributed System Security Symposium (NDSS'15)*, San Diego, CA, USA, Feb. 2015.

[65] Python Package Index. Pyside 1.2.4. `https://pypi.python.org/pypi/PySide/1.2.4`.

[66] A. Rahimian. BinSourcerer. GitHub project. `https://github.com/BinSigma/BinSourcerer`.

[67] A. Rahimian, P. Charland, S. Preda, and M. Debbabi. RESource: A framework for online matching of assembly with open source code. In *Foundations and Practice of Security (FPS'13)*, La Rochelle, France, Oct. 2013.

[68] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Symposium on Principles of Programming Languages (POPL'99)*, San Antonio, TX, USA, Jan. 1999.

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

[70] P. V. Sabanal and M. V. Yason. Reversing C++. In *BlackHat Security Conference*, Las Vegas, NV, USA, Aug. 2007. http://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf.

[71] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis (ISSTA'09)*, Chicago, IL, USA, July 2009.

[72] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*, Washington, D.C., Aug. 2013.

[73] Sirmabus. Class informer plugin. SourceForge project. http://sourceforge.net/projects/classinformer/.

[74] A. Slowinska, T. Stancescu, and H. Bos. DDE: Dynamic data structure excavation. In *Asia-pacific Workshop on Systems (APSys'10)*, New Delhi, India, Aug. 2010.

[75] M. Sojer and J. Henkel. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *Journal of the Association for Information Systems*, 11(12):868–901, Mar. 2010.

[76] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security (ICISS'08)*. Hyderabad, India, July 2008.

[77] Y. Srikant and P. Shankar. *The compiler design handbook: optimizations and machine code generation.* CRC Press, 2007.

[78] TIOBE Software BV. Tiobe index. Online article. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[79] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Working Conference on Reverse Engineering (WCRE'05)*, Pittsburgh, PA, USA, Nov. 2005.

[80] M. Van Emmerik. *Static Single Assignment for Decompilation.* PhD thesis, University of Queensland, 2007.

[81] K. Yakdan, S. Eschweiler, and E. Gerhards-Padilla. Recompile: A decompilation framework for static analysis of binaries. In *Malicious and Unwanted Software: "The Americas" (MALWARE'13)*, Oct. 2013.

[82] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Network and Distributed System Security Symposium (NDSS'12*, San Diego, CA, USA, Feb. 2015.

[83] F. Yamaguchi. Joern. GitHub Project. https://github.com/fabsx00/joern.

[84] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May. 2014.

[85] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Annual Computer Security Applications Conference (ACSAC'12)*, Orlando, FL, USA, Dec. 2012.

[86] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *ACM Conference on Computer and Communications Security (CCS'13)*, Berlin, Germany, Nov. 2013.

[87] Z. Zihui. RTTI technology and dynamic creation in MFC. *Microcomputer Information*, 9:79, Sep. 2008.

[88] Zynamics. Bindiff. Online article. http://www.zynamics.com/bindiff.html.