Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Jacob Hofgaard

Static control flow graph analysis for inlined function detection and identification

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
Co-supervisor: Solveig Bruvoll
June 2024

NTNU
Norwegian University of
Science and Technology

Jacob Hofgaard

Static control flow graph analysis for inlined function detection and identification

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
Co-supervisor: Solveig Bruvoll
June 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Abstract

In recent years, understanding the inner workings and components of software has gained significant importance due to the increasing number, size and complexity of programs. Understanding the low-level details of a program is important in malware analysis and developing low-level components like firmware. The complexity in programs is often a result of advanced compiler optimizations. Optimizations can produce executable code that appears different in structure from its original source. As software systems scale up, manual analysis becomes more challenging and time-consuming, highlighting the need for automated tools to assist in reverse engineering. Automated software analysis is essential for efficiently detecting and identifying pre-analyzed or documented code.

This project explores the use of control flow graphs (CFGs) to detect duplicate code and recognize code from libraries, focusing on identifying code affected by the compiler optimization technique known as function inlining. It also explores the verification and testing process of automated analysis tools related to function identification. The findings demonstrate that while control flow graph analysis is an effective method for detecting and identifying inlined functions, it remains computationally expensive.

Sammendrag

De siste år har forståelse for programvares indre funksjoner og komponenter blitt stadig viktigere. Dette er på grunn av det økende antallet, størrelse og kompleksiteten i programmer. Forståelse for de mer detaljerte nivåene i et program er viktig innen analyse av skadevare og utvikling av lavnivå programmer som fastvare, firmware. Kompleksiteten i programmer er ofte et resultat av avanserte kompilatoroptimaliseringer som kan produsere programmer med kode der strukturen er vesentlig annerledes enn i den opprinnelige kildekoden. Når programvaresystemer blir større og mer omfattende, blir manuell analyse mer utfordrende og tidkrevende, noe som understreker viktigheten av automatiserte verktøy for å bistå i omvendt utvikling. Automatisering av programvareanalyse er avgjørende for effektivt å oppdage og identifisere forhåndsanalysert eller allerede dokumentert kode.

Dette prosjektet utforsker bruken av kontrollflytgrafer for å oppdage duplikatkode og gjenkjenne kode fra biblioteker, med spesielt fokus på å identifisere kode påvirket av kompilatoroptimaliseringsteknikken kjent som funksjons inlining. Prosjektet utforsker også hvordan verifikasjon og testing av automatiserte analyseverktøy knyttet til funksjons identifisering kan gjennomføres. Funnene viser at analyse av kontrollflytgrafer er en effektiv metode for å oppdage og identifisere inlinede funksjoner, men det er en beregningsmessig kostbar teknikk.

Contents

Figures

Tables

Code Listings

Acronyms

AST Abstract Syntax Tree. 8

CFG Control Flow Graph. 23

CPU Central Processing Unit. 6

DLL Dynamic Linked Library. 11

ELF Executable and Linkable Format. 11

GCC GNU Compiler Collection. 20

IR Intermediate Representation. 9

LTCG Link-time Code Generation. 10

OS Operating systems. 6

PE Portable Executable. 11

Chapter 1

Introduction

Computers and software have become essential in daily life. Society has become dependent on computers and computer programs in both professional and private settings. Computers of all sizes, from smart devices to supercomputers, have been integrated into everyday activities. This has transformed how we interact and connect, whether for work, communication, entertainment, or managing personal affairs. The digitalization process has created opportunities and enhanced efficiency and connectivity.

However, increased dependence on digital systems has also introduced challenges and new vulnerabilities. Computers have become increasingly complex systems with components poorly understood by the general public. These systems are susceptible to exploitation by malicious actors working to achieve their own goals. By hiding in and abusing opaque and obscure components of a system, a malicious actor has the potential to do tremendous damage. This threat has been apparent both in personal and commercial settings. For example, the 2015 breach of the dating website Ashley Maddison exposed the private lives of millions of users [1], while one of Europe's largest shipping companies, Maersk, lost around $300 million because of a cyber attack in 2017 [2]. These are just some examples of the situations caused by dangerous cyber actors in the last years. The escalating threat of cyber attacks highlights the need for development, research, and optimizations in the field of cybersecurity.

1.1 Topics covered

Understanding how computer programs operate, has become increasingly important. In recent years, vulnerabilities and malicious software have become more widespread. Vulnerabilities are bugs in a program that can be exploited to gain access to a system, while malicious software is a complex program used by an attacker. Both detecting vulnerabilities and understanding malware require deep insight into the inner workings of software systems. Getting this insight is time-consuming and difficult.

The process of analyzing and understanding an already existing product is called reverse engineering [3]. In computer science and information security contexts, reverse engineering is associated with analyzing an unknown program to understand its functionality, discover underlying principles, or repurpose its components. This technique plays a crucial role in various fields, such as software development, cybersecurity, and hardware analysis. It enables professionals to deconstruct complex systems, diagnose vulnerabilities, and enhance existing designs.

Not all programs are easily analyzed. This can be because a program lacks proper documentation, has an unknown origin, or is even designed to make it difficult to reverse engineer. A common example of a program that is difficult to analyze and understand is malicious software. Malicious software is designed not to be detected, and if detected, its goal is to make it hard to analyze. Understanding malicious software is a large part of improving cyber security.

1.2 Keywords

Control flow graph, static analysis, reverse engineering, function identification, function inlining

1.3 Problem description

With the increased digitization, the need for software has increased, and efficiency in the development process is highly valued. Reusing code already developed by others can be time-saving when developing a program. This is often done by using libraries. Libraries are code developed to be generic and provide a desired functionality. They are designed to be reused by other developers. Code from libraries is already documented and can be used in all types of projects, from professional tools to malicious software.

The use of libraries also affects reverse engineering. Since libraries are already documented, there is no need to analyze them further. Detecting which part of a program is from a library is challenging, especially if the program being analyzed is compiled. A compiled program is a program converted from human-readable code, source code, into binary code understood by a computer. If a reverse engineer spends time analyzing something that is already documented, much work will be superfluous. If something is already understood and documented, it is unnecessary to continue analyzing it.

Optimizations in reverse engineering are crucial for improving efficiency, accuracy, and scalability. As systems become increasingly sophisticated and large-scale, the challenges associated with reverse engineering escalate. Effective optimization techniques can significantly reduce the time security researchers spend understanding malicious software and make it easier for developers to understand and enhance existing undocumented systems. One possible technique that would

improve the efficiency of the reverse engineering process is the identification of already documented functions, like library functions.

Identification of library functions in compiled programs is not always straightforward. When a program is converted from source code into a program, it goes through optimization steps. The optimizations done to a program are not always identical. It depends on the code context to determine what optimizations are effective. This makes it hard to identify a library function as it can take different shapes depending on how it is optimized. One standard optimization technique the compiler uses is function inlining. Function inlining is when a compiler replaces a call to one function with the content of the function. This process optimizes the execution speed of the program; however, it makes the program more difficult to analyze for someone doing reverse engineering.

Detecting and identifying library functions that have been inlined in a compiled program would decrease the time spent in analyzing already documented code in reverse engineering. There are multiple methods that could be used to detect and identify code; one of those methods is the use of control flow graphs. A control flow graph is a graph portraying the execution flow of a program. By analyzing this graph, it can be possible to detect structures that are inlined versions of library functions.

1.4 Research questions

The motivation for this master thesis is to find a technique to supplement the current initial analysis tools available when analyzing a completely unknown software program. Creating a generic method for identifying known code could save a lot of time. It would also give the reverse engineer a better opportunity to quickly assess a program's capabilities when reverse engineering new software. A lot of undocumented code, like malware, is written in compiled languages. Tools designed for this type of reverse engineering would, therefore, be disproportionately beneficial in security-related reverse engineering.

The thesis aims to explore how control flow graphs can be used to detect inlined functions. To accomplish this, it will explore the current research on function detection and identification, and determine whether control flow graph analysis can be used as a supplement. It will also provide a testing framework to decide if an analysis technique is suitable to be used in reverse engineering. The thesis will not provide an exhaustive evaluation but will provide a framework for future work.

Research questions:

1. Is control flow graph analysis an effective technique for identifying inlined functions?
2. What are the essential components and design considerations for developing an effective testing framework for inline function identification?

1.5 Contributions

This thesis has examined a technique for identifying known, documented code used within an unknown arbitrary program. This technique uses control flow graph analysis to detect and identify inlined functions. The effectiveness of control flow graph analysis as a technique for inlined function detection and identification has been tested. The tests showed that control flow graph analysis can detect and identify inlined functions. However, it has some limitations. These limitations relate to function detection in specific function types and performance. Furthermore, the thesis discusses situations where this method could provide an advantage and how it can be used on a larger scale.

These contributions offer insights into understanding possible techniques for supplementing the reverse engineering analysis processes. The work done in this thesis will extend the current understanding of control flow graph analysis in a reverse engineering context, and its strengths and weaknesses.

Chapter 2

Theory

Software reverse engineering is the process of analyzing and understanding a program. The following chapter explains the theory relevant to understanding the reverse engineering process of a binary program. The chapter describes the different parts of creating and running an executable binary program. An overview of the parts, the source code, compilation and linking, the executable file, and program execution are shown in Figure 2.1. All the steps, "Source code", "Compilation and linking", and "Executable files" and their execution, are described in Sections 2.1 to 2.3. Then, the compilation and linking process and how it optimizes a program is described in more depth. The chapter then explores the theory relevant to reverse engineering of binary programs. Finally, it discusses control flow graphs and how they relate and can be used to assist in reverse engineering.

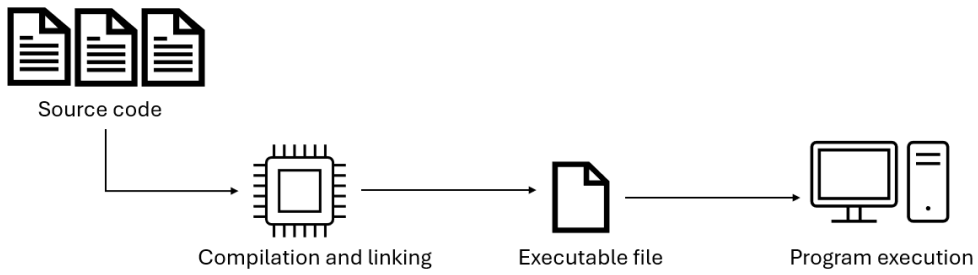


Figure 2.1: The figure shows a simplified overview of how source code is converted into a program and executed on a computer. The *compilation and linking*-step is expanded upon later in the thesis.

2.1 Source code

Source code is the input in creating an executable program as shown in Figure 2.1. Source code is a human-readable version of a computer program written in a programming language. The programming language defines how the source code is interpreted. Source code has different properties depending on the language it

is written in. Some languages need to be converted into machine code before they can be executed. This process is called compilation and is described in Section 2.2. Source code written in an interpreted language could be executed directly and does not need to be compiled. The difference between compiled languages and interpreted languages is explained further in Section 2.4

2.1.1 Open-source and closed-source software

One way to classify software is to differentiate between open-source and closed-source. Open-source software and closed-source software differ in their accessibility and transparency. For instance, Linux is a well-known example of open-source software where its source code is freely available for anyone to view, modify, and distribute. In contrast, Microsoft Windows operates as closed-source software, with its source code not accessible to the public. Open-source software allows anyone to modify, distribute, and use it freely. On the other hand, closed-source software is developed by a company or organization that keeps the source code proprietary. This means that users are not able to access or modify the code and must rely on the company to provide updates, new features, and bug fixes. While closed-source software often provides more polished and user-friendly products, open-source software offers greater flexibility and control for those who are comfortable with modifying a program themselves. Additionally, the open-source development model often leads to more rapid innovation and the ability for developers to collaborate and build upon each other's work.

2.2 Compilation and linking

The following section presents how a program is transformed from source code into a program that can run on a computer. The following section assumes the source code is written in a language like C or C++, which are languages that need to be converted from source code into a program before they can be executed. The main focus is on converting the program into machine code and not on how the program is handled when executed. It expands on the step called "*Compilation and linking*" in Figure 2.1 and explains the sub-parts this step is comprised of.

Executing a program on a computer is a multi-step process. The exact steps taken and how they are carried out depend on the Operating systems (OS), the Central Processing Unit (CPU), and the CPU architecture. These steps can be split into three main parts: the generation of the program, the loading of the program into memory, and the execution of the program [4]. Interpreted languages do not completely fit this model and are not relevant to this thesis. A program is generated in compiled languages when source code is compiled into machine code and given a structure that the OS can understand. The method a program is loaded and executed depends on the OS and is described in Section 2.3. This thesis focuses on the Windows OS on x86 architecture.

To generate a program that can run on a computer, source code written by a developer[1], needs to be converted from human-readable text into something that the computer can understand. This process is done in multiple steps. There are different ways to split up this process, but one way is described by Alfred V *et al.* [5]. The steps in Alfred V *et al.*'s model are listed below and is further explained in Sections 2.2.1 to 2.2.3.

1. Preprocessor: Collect all relevant files and expand macros.
2. Compiler: Translate source code files into binary files containing machine code.
3. Linker: Combines binary files and links them together into an executable program.

Figure 2.2 shows an expanded version of Figure 2.1. The "*compilation and linking*" step in Figure 2.1 is split into the steps listed above. The figure shows what the components of the code generation process are and how they interact with each other.

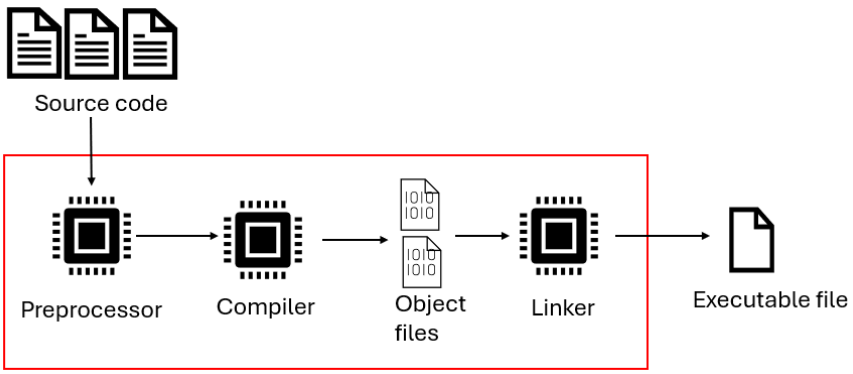


Figure 2.2: The figure is an expansion upon Figure 2.1. The *compiler and linker*-step is expanded into the steps in the model by Alfred V *et al.*[5]. The expanded steps are within the red frame.

The same program or component in a code generation toolchain can sometimes do several steps depending on the toolchain used. This makes the naming and separating of these steps less clear than visualized in Figure 2.2. One example of this is the preprocessor and the compiler. Most code generation toolchains implement the preprocessor and compiler in the same process. In addition, the whole process of generating an executable from source code is sometimes called *'compiling a program'*. In this thesis, the generation of a program is referred to as *'compiling a program'* since it is the established practice. If specific parts of the code generation toolchain need to be referred to, the name of that component

[1]Source code can also be generated programmatically by another program, but the next steps in the code generation process do not depend on how the source code is created. This thesis does, therefore, not differentiate between how source code is generated.

will be used, i.e., the linker, the compiler, or the assembler. The compiler and the linker components are the main focus of this thesis.

2.2.1 Preprocessor

The preprocessor is the first component that handles source code. As shown in Figure 2.2 it takes source code directly as its input. The preprocessor is responsible for collecting all source code files and expanding macros in the code. The text in the source code is converted into tokens. Tokens are text close to the source code, but ambiguities, like whitespaces, are removed to make handling the code easier for the following steps. These tokens are passed as output from the preprocessor and used in the following steps. The preprocessor is also responsible for collecting files referenced in the source code that are not directly part of the project, like expanding `#include` statements [6].

2.2.2 Compilers

The compiler is the next step in Figure 2.2. Compilers are computer programs that read tokens from a program written in one language and translate them into a program written in a different one. This is normally in the form of translating source code, written in a compilable programming language, into machine code or executable code that a computer can understand and execute [5]. Compilers are more complex than preprocessors and can be split into multiple parts. The parts may differ from compiler to compiler, but most compilers have similar functionality in their parts, even if they use alternative naming. The specific steps described below are based on the naming in the Clang/LLVM compiler [7]. These steps are:

1. Parsing and Semantic Analysis
2. Code Generation and Optimization
3. Assembler

Parsing and Semantic Analysis

This takes the tokens generated by the preprocessor and generates a parse tree called the Abstract Syntax Tree (AST). If the AST can not be formed, this generates errors. The AST is then analyzed to identify if the code is 'correct' or if it includes code that has some potential errors. This stage is responsible for generating the most common warnings seen during the compilation process [7].

Code Generation and Optimization

This step is responsible for converting the AST into something closer to code understood by a computer. The *Code Generation and Optimization* step often includes the assembler. However, it will explained separately below as it could be considered an individual component. An intermediate code is generated from the

AST during this step. The intermediate code, also known as Intermediate Representation (IR), is a hardware-independent low-level type language. Low-level languages is explained in Section 2.4. The AST and the IR are optimized during this step [8]. What properties are prioritized during the optimization step depends on the arguments given to the compiler process. Compiler optimization is expanded upon in Section 2.5. The output from this step is assembly code unless the *Assembler* is part of this step.

Assembler

The assembler takes the output from the previous step and generates machine code from the assembly instructions. This step is often done as part of the *Code Generation and Optimization* step. The output from this step is object files. Object files are files containing machine code. These files are not able to be run directly on the computer since they have no reference to where other parts of the code are located. Code from different source code files are put in different object files. These have labels or symbols that the linker uses to link different object files into a single executable [9].

Compiler keywords

Programmers have the ability to influence the compiler's behavior by incorporating specific keywords into the source code. These keywords, often called compiler directives or pragmas, provide explicit instructions to the compiler on how to handle certain portions of the code. By using keywords, programmers can optimize performance and control aspects of the compilation process. An example of this is the keyword `__always_inline`, which specifies that a function should be inlined [7].

Compiler flags

Compiler flags are settings or options passed to a compiler. These flags modify the compiler's behavior and influence various aspects of the compilation process, such as optimization level, debugging information generation, target architecture, and more. The specific flags and their meanings may vary depending on the compiler being used and the target platform. Different flags can significantly impact the compiled code's performance, size, and behavior. Since the compiler and linker are usually run together, some compiler flags are passed to the linker and affect the behavior of the linker as well.

Compiler flags are global and affect the general "behavior and priority" of the compiler. They are passed as arguments into the compiler process. In contrast to compiler keywords, compiler flags affect the whole compilation process and not specific functions or parts of the code. Flags and keywords are discussed further as part of code optimizations in Section 2.5.

2.2.3 Linkers

The linker is the final part in the *compilation and linking* process shown in Figure 2.2. It takes the output of a compiler, links the different parts together, and combines this with information the OS needs during the loading and execution of a program to make an executable binary.

A compiler generates machine code for the different source code files and creates separate files containing the machine code for each source code file. When code or data in other files are being referred to, the compiler creates a symbol to indicate what is being referred to. The linker is responsible for combining the different files generated by the compiler and replacing the symbols referencing other files with pointers to the correct location. Traditionally, the linker does not optimize and does not affect the code flow. However, compiler flags like Link-time Code Generation (LTCG) enable code generation during the linker step of compilation [10]. This flag enables the linker to modify the code and, therefore, do optimizations. This could include doing inlining across multiple different objects generated by the compiler, as is expanded upon later.

2.2.4 Compiler identification

When compilers and linkers generate a program from source code, they add artifacts specific to that compiler setup. These artifacts may be used to identify which compilation and linking tools were used. This can be done by using rule or signature-based detection techniques as described in [11]. Neural networks have also been used to try to identify compilers. According to Tian *et al.* [12] neural networks were able to identify the compiler, compiler version, and compiler optimization level with a high degree of accuracy.

Programs compiled using the Microsoft toolchain, i.e., the Visual Studio compiler, have a signature specifying the compiler and its version [13]. This signature is in the form of an extra header at the beginning of the executable, located between the DOS stub and the PE header[2]. It is called the RICH header. The RICH header is an undocumented structure created by Microsoft that has been reverse engineered [15]. It can be used to get information about the compiler. It has also been used in malware identification and classification [16].

2.3 Executable programs in operating systems

The operating system serves as the intermediary between the hardware and software of a computer. They provide an interface that abstracts the hardware, making it easier for software or applications to function. Programs running on an OS are contained in files known as executable files. Executable files are created when source code is compiled into a program. These files contain machine code that the

[2]More information about the DOS and PE headers can be found in [14]. It will also be explained in Section 2.3.1.

OS can directly execute. In this thesis, 'executable files' does specifically refer to files containing machine code in a structured format, such as Portable Executable (PE) and Executable and Linkable Format (ELF) files. Scripts or other files that can be executed are referred to as 'scripts' or 'interpreted code'. Executable files contain machine code meant to be run on an OS in a predefined format. These files contain the machine code instructions and information specifying how the execution environment should be set up and executed by the OS. This section explores the structure and characteristics of executable files, focusing on the two most common formats: Windows PE files and Linux ELF files.

2.3.1 Portable Executable

The PE format is a file format used for executable files and Dynamic Linked Library (DLL) in the Windows OS. The PE format is widely used for Windows applications. Understanding the PE structure is crucial for reverse engineering, malware analysis, debugging, and other security-related tasks. It provides a standard structure for the executable files and is designed to be independent of the underlying hardware platform; instead, it interacts with the OS. One example of this is the handling of memory. The PE structure and the program assume that no other programs are running on the system and all of the memory space is available. In reality, the OS handles the memory allocation and allocates physical memory to addresses unrelated to where the PE structure assumes it will be allocated [17].

The PE structure consists of a header and multiple sections containing code, data, and resources. The header contains information about the file, such as its size, entry point, and section layout [14]. The sections contain all the necessary data and instructions for running the executable. Not all header parts are needed, some are optional. Which optional headers are used depends on the code and the compiler compiling the code.

2.3.2 Executable and Linkable Format

ELF is the Linux equivalent of Windows's PE files. The ELF standard was developed in the 1990s [18], and adopted as the Linux standard binary format in 1999 [19]. ELF files contain most of the same information as the PE standard.

According to [20] and [21], Windows has around 70% of the market share in the desktop space, whilst Linux has around 4% of desktops, and macOS has about 15%. In the server space, this is flipped, and Linux has the majority share [22]. This thesis mainly focuses on the desktop environment, and therefore the Windows OS is the main focus.

2.4 Levels of abstraction

In software development and reverse engineering, levels of abstraction refer to the varying degrees of complexity and detail presented. These levels range from

high-level abstractions, which hide complex details and provide a user-friendly interface, to low-level abstractions, which expose the intricacies of the underlying hardware and system operations.

Different characteristics are desired from code depending on the perspective of someone looking at the code and where it is running. A programmer designing a web page has a different perspective than a firmware developer. To satisfy different perspectives, several layers of abstraction are used. This section gives a brief overview of these levels and expands on the levels relevant to the thesis. There are different ways to divide into these abstraction layers, but this thesis uses the six levels of abstraction described by Sikorski and Honig [23].

1. Hardware
2. Microcode
3. Machine code
4. Low-level languages
5. High-level languages
6. Interpreted languages

The abstraction levels can be placed in the context of the code generation process as shown in Figure 2.3. The figure shows how code written in the higher abstraction levels can be converted and end up as a program running on a computer on the lowest abstraction level.

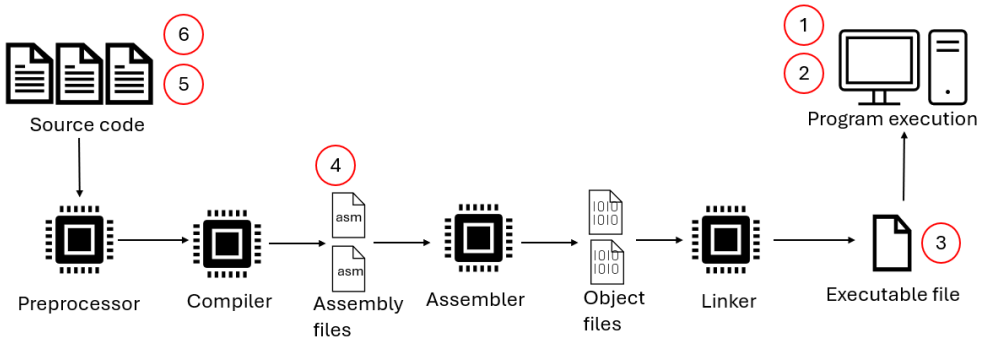


Figure 2.3: The levels of abstraction are shown in the context of the code generation process. The numbers in the figure correspond to the levels of abstraction, from 1: "Hardware" to 6: "Interpreted languages".

2.4.1 Overview of abstraction levels

This section gives a brief description of the abstraction levels. Section 2.4.2 and the following sections go more in-depth into the abstraction levels that are most relevant to the thesis.

Hardware

The hardware level is the electrical circuits inside the processor. This is the lowest level of abstraction, and all operations on higher levels are ultimately converted and executed at this level. Since this level is implemented in electrical circuits, it is not possible to modify it in software. The hardware level is applicable to all physical processors in a computer, but only the CPU is focused on in this thesis.

Different hardware processor units have different architectures. This is what is called CPU architecture. The most common processor architecture types are *x86* and *arm*. This thesis focuses on the *x86* architecture, which is currently the most common architecture for PCs [24].

The CPU is the main component that executes instructions. It includes components to fetch data and new instructions, execute an instruction, and several registers containing the most immediate data needed for executing the next instructions. These registers store data and memory pointers, i.e. values and pointers to values. They also store flags related to the current state of the CPU. These flags are called CPU flags and are stored in a dedicated register. CPU flags are used in conditional and branching operations. They are described in Section 2.4.2.

Microcode

Microcode (sometimes written as µcode) is the interface between hardware and machine code, as described in [23, p. 66].

> ”The microcode level is also known as *firmware*. Microcode operates only on the exact circuitry for which it was designed. It contains microcode instructions that translate from higher machine code levels to provide a way to interface with the hardware. ” - Sikorski and Honig [23, p. 66]

Because of the connection with the hardware level, this is not usually a level most programmers or reverse engineers need to examine. However, the microcode level is important for understanding low-level CPU optimization techniques, like Out-Of-Order execution [25], and has introduced vulnerabilities like Specter [26] and Meltdown [27].

Machine code

Machine code is the bytes the CPU reads from memory, interprets, and then executes. Different CPU architectures could interpret the same bytes as different instructions, so machine code is highly connected to the platform it is running on. The CPU interpreting machine code does not need to be physical; it could be virtual.

Execution of a single machine code instruction could be converted into several microcode operations depending on the instruction and the microcode architecture. For example, the machine code instruction `ADD EAX, EBX` will generate microcode operations related to the addition of the two numbers and the setting of

affected CPU flags. Whereas the instruction `ADD EAX, [EBX]` will generate more microcode instructions, additional instructions are needed for retrieving the value at the memory address pointed to by `EBX`, as well as the ones for actually doing the addition [25].

Low-level languages

Low-level languages are human-readable versions of machine code. The most well-known is Assembly. The language is a conversion of the machine code with a few additions to make it more human-readable and user-friendly. This is further discussed in Section 2.4.2.

High-level languages

High-level languages have no direct connection with the machine code or lower levels. Figure 2.2 shows how source code written in a high-level language is converted into machine code. To run a program written in a high-level language, the program needs to be converted into machine code; this topic was explored in Section 2.2. Humans find high-level languages a lot easier to use since they remove several of the details needed in low-level languages. For example, in high-level languages, the programmer does not need to handle memory allocation of the stack. Another example is that a single instruction in a high-level language will likely be converted into multiple machine code instructions, similar to how a machine code instruction is converted into multiple microcode instructions.

Interpreted languages

Interpreted languages are languages where a program is written and not converted into machine code before it is run. Instead, an interpreter converts the program during run-time. This can be done in multiple ways: using an interpreter to interpret and execute instructions continuously or doing a compilation-like process during the program's runtime. Compilation during runtime is called Just-In-Time (JIT) compilation, and it is used as an optimization step. The JIT compilation does not necessarily compile code into machine code understood by the CPU. It could instead be compiled into a machine code-like language that runs in a custom virtual machine.

2.4.2 Assembly

Assembly is the human-readable version of machine code and a low-level language[3]. Each "line" in an assembly file represents a separate machine code instruction. Each machine code instruction is fetched and executed independently

[3]Some sources, like [28] do not differentiate between machine code and assembly but describe "assembly as a symbolic representation of machine code." In this thesis, assembly is the human-readable version and machine code is the binary representation

by the processor. The naming convention of assembly instruction differs depending on the sources and architectures used. This thesis uses the *Intel x86/x86-64* naming convention and syntax, as shown in Code listing 2.1. There exists other assembly syntax, like the AT&T syntax. This thesis uses Intel syntax because most tools used in this thesis also use the Intel syntax. More information about the differences between AT&T and Intel syntax can be found in [29] and [30].

Code listing 2.1: Intel format of assembly instructions

```
LABEL: MOV EAX, 0x1234
label: mnemonic operand1, operand2
```

In some settings, it is necessary to be able to look at the machine code and the assembly of an instruction. An example of the format for this is shown in Code listing 2.2. The first part, the sequence "*BA 0F 00 00 00*" in the example, is the machine code of the instruction. The sequence contains numbers in hexadecimal. The last part is the assembly in the Intel syntax described above.

Code listing 2.2: Machine code and assembly of a mov instruction

```
BA 0F 00 00 00               mov     edx, 0Fh
```

As shown in Code listing 2.1, each assembly instruction comprises up to three parts. Not all instructions use all parts. These parts are:

1. Label
2. Mnemonic
3. Operand

Label

Labels are used as references in the code. The label is a marker added to make the code easier to read and is not part of the machine code. In machine code, memory addresses are used directly if there is a reference. Labels are a symbolic (human-readable) representation of that memory address. Not all assembly instructions have labels. Labels are usually only present if some other code references the address of the instruction or if it improves readability. The labels themselves do not contribute to the machine code and are only present if a programmer writing assembly adds them or they are added by a disassembler.

Mnemonic

Mnemonics is the name of the operation that will be performed when an instruction is executed, as shown in Code listing 2.1. In Code listing 2.3 `jne` is the mnemonic in that line. Mnemonics can also be described as the name of the operation being done in an assembly instruction.

Mnemonics does not always fully describe exactly what is being executed; this is further explained in the following paragraphs. Because of the ambiguity in some mnemonics, there exists a more precise way to explain what an instruction

does. In this paper, this is referred to as the 'opcode'. The opcode is the first part, mnemonic equivalent, of the instruction in machine code[4], i.e., in Code listing 2.3, '75' is the opcode of the conditional jump instruction.

Code listing 2.3: Dissasembly of a conditional jump

```
75 45                   jne     label_1
```

The same mnemonic sometimes refers to different opcodes. This can occur because of the operands used in the instruction, as shown in Code listing 2.4. This depends on the operands in the instruction. In assembly the operand type is not necessary to specify in the instruction, but this is needed in machine code.

Code listing 2.4: Variants of move instructions with different opcodes

```
C7 45 D8 00 00 00 00    mov     [ebp-40], 0
89 45 A4                mov     [ebp-92], eax
8B 10                   mov     edx, [eax]
```

Sometimes, certain keywords are added to the assembly instruction to add specificity, like `short` in Code listing 2.5. The addition of `short` in the listing specifies that the conditional jump is relative and to an address a 'short' offset, a distance possible to specify with a single byte as offset, from the current instruction pointer.

Code listing 2.5: Same mnemonic with different opcodes

```
0F 85 B9 00 00 00       jnz     label_1
75 0D                   jnz     short label_2
```

Both instructions in Code listing 2.5 are conditional jumps, and will change the instruction pointer if the *zero-flag*, one of the CPU-flags, is 0. Information about different conditional instructions and CPU flags can be found in [28, p. 3-16]. CPU flags are affected by the instructions shown in [31], but it is not directly relevant to the thesis and is not be further explained. The mnemonic for both instructions in Code listing 2.5 is "jnz", however, the opcodes differ. The first conditional jump is a jump to an address specified by a 32-bit[5] offset relative to the current address. The second conditional jump uses an 8-bit offset relative to the current address instead of 32-bit.

Some mnemonics are synonyms and refer to the same opcode. This is common in conditional operations. In Code listing 2.6, the two mnemonics, `jne` and `jnz`, refer to the same opcode and can be used interchangeably. This is done to improve readability. From the CPUs perspective, the same CPU flag is set if a comparison equals zero and if two equal values are compared [28], [33]. Not all similar-sounding mnemonics are synonyms. For example, "greater" and "above" are used

[4]Some sources define the opcode as the whole machine code instruction. In this thesis, the whole instruction is referred to as 'machine code of an instruction' or just 'machine code'. The opcode is only the mnemonic equivalent of the machine code instruction.

[5]Assuming the CPU is in protected(32-bit or 64-bit) mode, not real mode. Real mode is not relevant to the thesis, and all code and examples are in protected mode. More information about real and protected mode can be found in [32].

to indicate a signed or unsigned comparison and can not be used interchangeably. The full list of synonyms for conditional mnemonics can be found in [34].

Code listing 2.6: Different mnemonic with same opcode

```
75 45               jne     label_1
75 45               jnz     label_2
```

Operand

Operands are the values given as input and output in a mnemonic and could be seen as somewhat equivalent to function arguments. Some instructions do not take operands, and some take several. Operands can be different types, and different instructions. Operands can be values, registers, or pointers to an address in memory. In the Intel assembly format, operations with an output or destination where the output or results are saved normally, have the format `"mnemonic destination, source"`, as described in [28, p. 1-7].

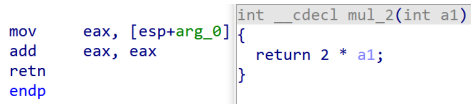
Some operations have effects that are not directly linked to the operands. Most instructions affect CPU flags in some way. This is further described in [31]. In most normal programs comparison instructions like `CMP` and `TEST` are used to affect the control flow of a program. These instructions directly alter CPU flags, which control branching operations. However, other instructions, like the `MUL`-operation, also affect general-purpose registers. `MUL` only has a single input operand, but the result is outputed in multiple registers, `EDX:EAX` in an x86-architecture CPU.

2.5 Code optimization

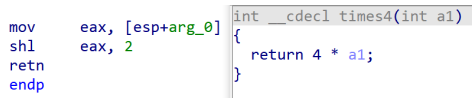
When source code is converted into machine code, the compiler aims to optimize the program being created. This optimization is done to enhance performance, minimize size, or improve other desirable characteristics. The code that is best for humans to read and understand may not be optimal for a computer when directly converted into machine code. Humans often write code for human readability, which is not essential to a computer. This can result in unoptimized code from a computer's perspective. To bridge this gap, compilers attempt to optimize the code they generate. The two most common characteristics that compilers optimize for are speed and size. The choice of optimization depends on what the developer specifies the compiler to optimize for when compiling the program. These two optimizations can conflict with each other, and the compiler prioritizes one based on the optimization flags used. For instance, optimizing for speed may result in a larger code size, while optimizing for size may lead to slower execution. The developer needs to consider these trade-offs when setting the optimization flags.

One way a compiler optimizes code is by replacing one instruction with another. Certain instructions might be faster or smaller in size, compared to others and are therefore preferred by the compiler. One example of this is shown in Figure 2.4. Multiplication is an expensive operation compared to other operations

[35]. As shown in the figure the compiler will replace multiplication with faster instructions if it is possible. `a1 * 2` is replaced with `a1 + a1` since additions are faster than multiplication. `a1 * 4` is replaced with shifting the value to the left, `a1 << 2`. Instruction optimization tries to replace slow instructions, instructions that take a lot of CPU cycles to complete, with faster instructions. Instructions can also be replaced with instructions that make execution prediction easier. Execution prediction is also called *speculative execution*[6] by Intel. This type of optimization does not change how the program is structured, even though the instructions differ from what the programmer might expect from the source code. Some other types of code optimization do, however, affect the structure and control flow of a program.



(a) Multiplication (a*2) optimized by replacing multiplication with addition using the add-instruction



(b) Multiplication (a*4) optimized by replacing multiplication with left shifting using the shl-instruction

Figure 2.4: The figure shows how the compiler uses addition and shift instructions to speed up multiplication. Figure 2.4a shows how a multiplication with '2' is replaced with addition. Figure 2.4b shows how the shift operation is used instead of multiplication.

The compiler could also remove code that is unused or not accessed. Larger unstructured projects could contain legacy code that is no longer referenced; if the compiler discovers this, it could remove the code. This would decrease the size of the program and make execution prediction easier. This type of optimization is called *dead code* removal.

2.5.1 Inlining

Inlining is an optimization technique focusing on making a program faster. Inlining is when the compiler takes a call to a function and replaces the call with the content of the function that was intended to be called. An example of this is shown in Figure 2.5.

This removes the need for a function call since the called function is put inside as part of the calling function. From an optimization perspective, this improves the performance of the program. Since no call is made, there is no need to establish and set up a stack frame, or remove it afterward. Additionally, it is not necessary to set up arguments in the same way as when a function is called. There is also an improvement in caching and pre-fetching. The instruction pointer does not jump to a new location. It is more likely that the next instruction is part of the current memory page and, therefore, cached.

[6] Speculative execution and fewer cache-misses or cache optimization is not relevant to this thesis, more information about this topic can be found in [36].

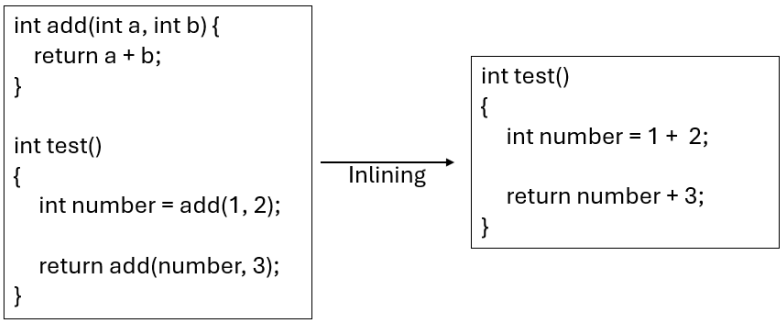


Figure 2.5: The figure shows how the function "*add*" is inlined into the function *test*. The content of the *add* function is inserted into where a call to the function was.

Because inlined functions originally were normal functions, some of the same attributes apply to them, with some modifications:

Single entry point

Normal functions only have one "entry point". Any normal function will always be entered at its beginning. This is to make sure that it is able to set up its stack, parse arguments, and save any non-volatile registers[7]. In theory, there is nothing stopping a program from jumping or calling any memory address, i.e. not the function "entry point", if it contains executable code, but this is not normal behavior, and because of this, it is ignored.

Inlined functions also, in a way, have a single entry point. Since an inlined function is a function that has been "copied" from one place to another, by the linker or compiler, it originally had a single "entry point". However, since it isn't its own function the stack setup isn't needed, and there isn't a clear calling convention defining how arguments should be passed to it. The optimization steps during compilation could change the layout of the start of the function. Parts of the function could also be removed by the optimization process.

Return to caller

A function has to return to the place it was called from. This also somewhat applies to inlined functions. An inlined function is not directly called, but as described above, it is originally a callable function. To preserve the code's functionality when inlining a function, the compiler has to ensure that the inlined function "returns" to the correct place and that the correct value is "returned". Since there isn't a return instruction and an inlined function is part of another function, the compiler

[7] Some registers, like the return register, 'RAX' in x86-64, may be overwritten by the callee, other registers must be preserved by the callee and returned in the same state to the caller. This depends on the calling convention. More information can be found in [37].

and linker will optimize it, as explained in the previous paragraph. This optimization could include modifying the control flow of the entire function, making the original function with the inlined function inside it appear differently than just the original function with the inlined function "copy-pasted" into it.

2.5.2 Inline optimization

If a compiler decides to inline a function depends on the source code, the compiler, and what flags are given to the compiler. Flags and keywords differ from compiler to compiler, but most compilers have options with the same meaning, but the names might differ. The following examples are from Microsoft Visual Studio.

As part of the code optimization step when compiling a program, the compiler tries to calculate the cost and benefit of inlining functions. The cost, usually larger code, is compared with the benefit, usually faster execution, and if some threshold is met the function is inlined.

Keywords in the source code like "`__inline`" or "`__forceinline`" incentivize the compiler to inline the function modified by the keyword [38]. These keywords affect the cost calculation described earlier and shift the threshold for the inlining of a function. The keywords are, therefore, more of a suggestion to the compiler than an instruction; even `__forceinline` does not guarantee the inlining of a function. Security considerations, recursion, and exception handling are some of the things that could stop a function from being inlined.

The flags given to the compiler affect how a compiler weights different alternatives during optimization and what optimization choices the compiler is allowed to make. The different compilers have different options for inlining, as shown in Table 2.1. The flags shown in the figure are not the only flags affecting function inlining in the different compilers. Other flags, like `-finline-stringops` in GNU Compiler Collection (GCC), affecting the inlining of string-related functions, also affect function inlining in general [39]. However, the primary compiler flags that enable and disable if the compiler is allowed to inline anything at all are listed in the table. Other inlining-related flags primarily affect what the compiler should prioritize or be limited by. The max stack size in a single function, the size of an inlined function, and the max instances of a function that can be inlined, can all stop a function from being inlined. How these flags affect the compiled program differs depending on the program being compiled. However, according to the documentation [7], [39], [40], these flags should not affect how an inlined function is structured or appears.

When reverse engineering a program there is no direct way to differentiate between a function being inlined because of the cost analysis or because the programmer specified certain keywords or flags. By looking at what is being inlined a reverse engineer could make some educated guesses, but there are no artifacts left in the program that give an unambiguous answer.

Table 2.1: The main compiler flags used to enable and disable function inlining for a selection of compilers.

Compiler	Enable function inlining	Disable function inlining
GCC	`-finline-functions`	`-fno-inline`
clang	`-finline-functions`	`-fno-inline-functions`
msbuild & cl	`/Ob3`	`/Obd`

2.5.3 Outlining

The opposite of inlining is outlining. This is when a part of a function is taken out and put into its own function. Outlining could reduce the size of a program if the code block that is being outlined is used in multiple locations in the program. This is a new addition to compilers compared to inlining. Outlining was added to LLVM in 2017 [41]. Outlining, in general, does not make a program harder to reverse engineer since there is no need to manually recognize the parts of the program that are used in multiple places and, therefore, only need to be analyzed once.

An example of outlining can be seen in Figure 2.6. This figure shows the opposite case compared to Figure 2.5. Outlining is not normally done with code as small as shown in Figure 2.6, as it would not provide any decrease in the size of the program by outlining a single line of code. Since outlining does not negatively affect the difficulty of reverse engineering it is not discussed further in this thesis.

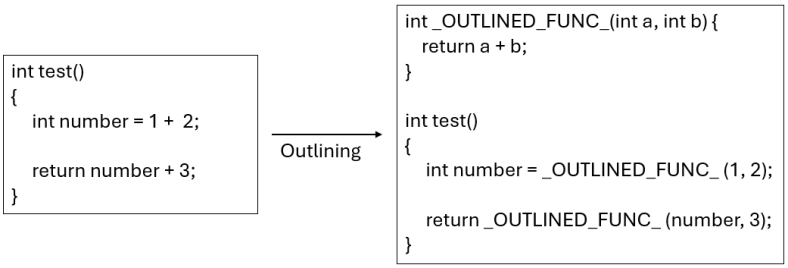


Figure 2.6: The figure shows the opposite of Figure 2.5. In the figure, the addition of two numbers is outlined from the function *test* into a function called *_OUTLINED_FUNC_*.

2.6 Reversing the compilation process

Analyzing a compiled program in its binary form, machine code, is a non-viable option. Therefore, tools designed to convert it from machine code to a more human-readable format have been essential in reverse engineering. This is done in two processes called *disassembly* and *decompilation*. Figure 2.7 shows how the two processes relate to the code generation process. Disassembly and decompilation are explained in the following sections.

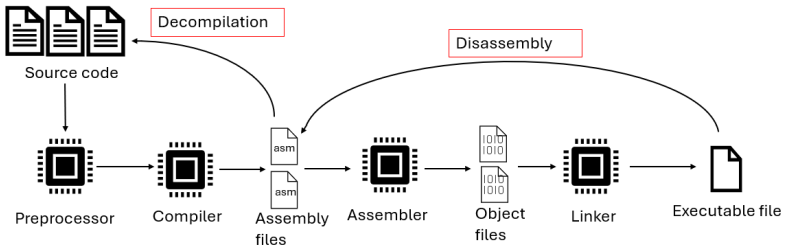


Figure 2.7: The figure shows how the compilation process can be reversed from machine code in an executable to low-level languages with disassembly and from low-level languages to high-level languages with decompilation.

2.6.1 Disassembly

The basic process of disassembly is converting machine code into assembly code. This is done by taking the given machine code, looking up what that series of bytes are as opcodes, and returning the equivalent assembly instructions. The architecture of the machine code and the attributes associated with the data currently being disassembled need to be a prerequisite to disassembling a program. Machine code does not have any intrinsic properties that make it possible to determine what instruction architecture it is. This is normally not a problem since most machine code is in a context, like an executable file or a library, which contains headers specifying the architecture and the properties of different sections of a program. There are examples of situations where this context could be missing, like shellcode, machine code injected into a process [42]. However, this is not relevant to the thesis.

Normally disassembly refers to the disassembling of machine code running on hardware, like machine code in x86-64. It could also refer to bytecode running on a virtual machine, like java bytecode or code obfuscated by VMProtect [43], [44].

Basic blocks

As described in [45], basic blocks are a series of machine code operations that will, from the perspective of a single thread, always be executed in order if not affected by any outside operations. A set of instructions without branches or conditional operations that will affect execution flow. Every basic block has to have a single "entry point", i.e., no jumps are allowed to the middle of a basic block. A basic block ends when one of the following situations occurs.

- The next instruction is an operation where the address of the following instruction is dependent on the state of the program.
- The next instruction is an "entry point" or the beginning of a new basic block.
- The next instruction affects the instruction pointer in a way where the next instruction to be executed is not the next in memory (like a jump or a return instruction).

The `CALL`-instruction does, however, not break up a basic block; this is because it is assumed that the program will return to the instruction after the call-instruction when the called function is done and returns. This assumption should be correct unless the called function doesn't return, like the function `ExitProcess` in the Windows API [46].

A `CALL`-instruction could also not return to its return address if the program does something unexpected like modifying its return address. The execution would then continue where the modified return address pointed to. This type of modification of control flow is most commonly associated with *Return Oriented Programming*, sometimes just called *ROP*. Return-oriented programming is normally an exploit technique used to get code execution if a buffer overflow occurs in the stack. It was invented as a response to the introduction of non-executable memory [47]. However, return oriented programming could also be used as a method to obfuscate the control flow of a program.

2.6.2 Decompilation

Disassembly is sometimes confused with decompilation. Decompilation is the process of trying to create a representation of the program in a high-level language from the compiled program [48]. This high-level representation is not necessarily the same as a source code equivalent of the program. When disassembling a program, information about the source code is not normally known. Information like what language the source code was written in could be undetectable. Some disassemblers like IDA PRO, therefore, always disassemble the program into a C/C++ or "C-equivalent" high-level language regardless of the language of the source code [49].

A part of decompilation is disassembly. By first disassembling the program an analysis tool is sometimes able to detect certain features or sets of instructions and interpret them as a specific high-level language feature, like loops, switches, or if-else statements. Certain nuances in the source code, like if a for-loop or a while-loop is used are seldom detected or differentiated in the machine code. This is not necessarily a weakness in the decompilation; nuances like that are sometimes removed by the compiling process and create situations where there are not any differences in the machine code.

2.7 Control flow graphs

A Control Flow Graph (CFG) is a graph created from the execution flow of a program. Each node in the graph is an operation or set of instructions the program executes, and each edge is the relations between each execution block [50]. CFGs have been proposed and used as a technique in program analysis for more than 50 years. CFGs are further expanded upon in the following paragraphs.

CFGs are used in reverse engineering to analyze a program's behavior, identify potential errors or inefficiencies, and optimize the program's performance. They

can also be used in debugging and dynamic analysis to trace the flow of execution. Malware detection and classification have been a field where CFG analysis has been used extensively during the last few years.

CFG generation and analysis are a somewhat expensive form of analysis. It is most commonly used in the classification and categorization of malware [51], [52]. In recent years research into combining machine learning and CFG analysis has also been used to identify and extract behavioral patterns from a malware sample using static analysis [53]. Different types of CFGs have also been used to analyze and classify malware, using code flow graphs and call graphs [54].

2.7.1 Control flow graph definition

A graph is a combination of a set of nodes and a set of edges. The total amount of nodes and edges is called a graph. Nodes are objects with one or more of the following: attributes, labels, and properties, depending on the data model. Edges connect the nodes. They must have a source node and a destination node [55]. An edge could have the same source and destination node. Edges are objects that can have different properties, like weights depending on the data model. An edge with a weight is called a *weighted edge*.

There are different types of graphs depending on what type of edges are used, *undirected edges* and *directed edges* [56]. *Undirected edges* are edges where the position of the source and destination node are irrelevant. In a graph with two nodes, A and B, and *undirected edges* an edge with source A and destination B will be equivalent to an edge with source B and destination A. In *directed edges* the position of the source and destination does matter, and graphs with directed edges can be used to form hierarchical structures. Both *undirected edges* and *directed edges* can be weighted.

A CFG is a graph representation of a computer program's control flow. A program's control flow determines the order in which it executes its statements and is influenced by conditional statements (like if-else or switch statements), loops (like for or while loops), and function calls.

In a CFG, each node represents a block of code, while the edges between the nodes represent the control flow between the blocks. The edges in a CFG are unweighted directed edges, meaning edges are not associated with any cost and the control flow in a program is directional. The nodes are usually labeled with some properties of that code block, like a signature, an address, or/and some set of instructions that the code block encompasses. The edges in a control flow graph can be labeled with the conditions that must be met for the flow to follow that particular edge [57].

2.7.2 Control flow graph types

A CFG can be created with different objects considered nodes and edges depending on the use case. Two examples of this are:

- a graph based on basic blocks
- a graph based on function calls

Basic blocks are a series of operations without branches or conditional operations as described in Section 2.6.1. A CFG based on basic blocks creates a node for each basic block and creates edges based on how these blocks are connected. This is a low-level graph where everything is created from the disassembled code. A call graph-based CFG is a graph created from the relations between different functions or subroutines [58]. Each node in the graph is a function, and each edge is an invocation of one function from another. A call graph could be created statically by disassembling a program and creating a graph based on each call instruction, or dynamically by running the program and tracing each function call.

A CFG created using static analysis could be incomplete since some edges could be generated dynamically. This happens if the address of a jump is calculated during runtime. Code branching based on runtime-determined addresses is called "indirect jumps" [59]. A common example of indirect jumps is in object-oriented languages. Object-oriented languages contain classes with functions or methods and data attached to each instance of a class. Methods in a class are often invoked based on a virtual function table specific for one instance of a class; a call graph-based CFG would, therefore, be hard to create without tracing the program.

A CFG can also be created dynamically by running the program. This type of CFG is produced by tracing the execution path of a program and marking the parts of code that are executed during runtime. While this approach can trace edges that are generated at runtime, it also has limitations. A dynamically created CFG is only a snapshot of the program's control flow during a specific run, and it may not capture all possible paths that the program could take. This can limit the extensiveness of the graph and potentially affect the accuracy of any analysis done using the graph.

CFGs are not intrinsically connected to compiled languages or code but could be created from interpreted languages. CFGs created from an interpreted language will have some challenges connected to them since it is not machine code that needs to be analyzed into a graph. Dynamically typed languages could also bring some unexpected situations since a function call in a dynamically typed language depends on the argument type to decide if a specific implementation of a function should be called, this could create more complex CFGs [60]. This thesis focuses on compiled languages; further research could, however, be done into testing if the same methods and techniques would be applicable to other language types.

2.7.3 CFG vulnerabilities

CFG generation is, like many other analysis techniques, vulnerable to obfuscation of the program that is being analyzed.

Opaque predicates is a technique specifically designed to obfuscate the control flow of a program by inserting false branches. False branches are branches where one path is always chosen [61]. This is achieved by creating branching operations where one branch always is taken. An example of an opaque predicate is shown in Code listing 2.7. Regardless of the value of `eax` the least significant bit will always be 0 when shifted 1 to the left [62], the `cmp`-instruction would therefore always set the CPU flag `ZF` and the execution would jump to the address of the label `ALWAYS_JUMP_HERE`. In CFGs, opaque predicates create false nodes and edges if they are not detected, which could, therefore, make the graph harder to analyze or negate the result of the analysis done on the CFG.

Code listing 2.7: Opaque predicate

```
shl eax, 1
cmp eax, 1
jz ALWAYS_JUMP_HERE
jmp NEVER_JUMP_HERE
```

Dead code insertion is another known obfuscation technique. "Dead code" is code in a program that doesn't affect the execution result, the state of the program is the same before and after the code has been executed. This could be achieved by inserting a single "nop"-instruction, or more complex operations as long as no data or program state has been modified after the execution of the dead code [63]. Dead code does not have to be an obfuscation technique but could be the result of poorly optimized code or legacy code that no longer is used [64].

2.7.4 CFG analysis implementations

CFG analysis has been implemented in several reverse engineering tools. It has a different understanding of a program's structure compared to other methods. This makes it suitable for comparing code. Methods comparing code byte for byte would be susceptible to assuming differences exist, even if the difference is only in a code's offset. A CFG-based analysis could track the different offsets and, more precisely, mark the actual deviation and not deviations attributed to variations in offsets. Two tools implementing CFG analysis are *Diaphora* and *Bindiff*.

Diaphora [65] is a plugin to IDA Pro[8]. This plugin is a collection of tools created for finding the difference between two compiled programs. In 2018 Karamitas and Kehagias [67] used Diaphora to find differences in functions based on the CFG of those functions. This was not a technique previously used in Diaphora and created a 2,5 times speed increase compared to the original version of Diaphora. It was equal in or had better accuracy. The results discovered by Karamitas and Kehagias

[8]IDA Pro is an "Interactive disassembler" created for disassembly, decompiling, and debugging software, commonly used in reverse engineering. It is developed by a company called HexRays [66].

show the potential gains of using CFG analysis as a binary analysis tool. A version of the method was later implemented in Diaphora [68]. Diaphora did not publish if their implementation of the method performed any differently than the one created by Karamitas and Kehagias.

Bindiff [69] is another tool created for comparing two programs. Bindiff has implemented several different comparison methods, some of which use control flow graph analysis. CFG is used when trying to match entire functions and in basic block matching. This is done by comparing edges as well as the nodes themselves in either a call graph-based analysis or a code flow graph. Bindiff is commonly used in finding differences when programs receive updates or are patched. In comparisons between Bindiff and Diaphora, Diaphora has been shown to be a bit more accurate [70].

2.8 Hash algorithms

Hash algorithms are algorithms that convert input data of arbitrary size to a fixed-size representation of the input. The output is a number called a hash value or digest. This hash value is supposed to be a unique representation of the input data. In theory, the hash of two different input data sets should never be equal for all data. In practice, this is not possible since the size of all possible input data is greater than the size of available hash space. Hash values are used for data integrity checks, data indexing, and data comparison. The first mention of hashing was in 1953 by Luhn [71] according to [72].

There are different types of hash algorithms, and they can be broadly classified into two categories: cryptographic and non-cryptographic [73]. Cryptographic hash algorithms are designed to be secure and irreversible, meaning that generating the same hash value for two different input data is almost impossible. Hash functions are one-way functions, and the only way to find the input used to generate a hash is to hash different inputs until a hash match is found. As cryptographic hashes are computationally expensive, this is a difficult task. These algorithms are used for digital signatures, password storage, and other security applications. Some popular cryptographic hash algorithms include SHA-256, SHA-1, and BLAKE2.

Non-cryptographic, or data-oriented, hash algorithms, on the other hand, are designed for speed and efficiency. These are typically used for data indexing, data comparison, and other non-security-related tasks. Some common non-cryptographic hash algorithms include MurmurHash, CityHash, and FNVHash [74].

Hash algorithms have become essential in computing and security. They provide a fast and efficient way to index and compare data, and they play a critical role in ensuring data integrity and security. In this thesis, the security and data integrity part of hashing is not focused on. Hashes are used to compare and index data.

2.8.1 SHA1

SHA1[9] is a hash algorithm published in RFC 3174 in 2001 by Eastlake and Jones [75]. SHA1 was a common cryptographic hash algorithm, but several weaknesses have been found. US National Institute of Standards and Technology has encouraged transitioning away from it in security applications [76]. These weaknesses do not apply to SHA1 as a data comparison algorithm. US National Institute of Standards and Technology still acknowledges it as an acceptable hash algorithm to use in non-cryptographic use cases, like data comparison where it is still quite common [77]. We used SHA1 in our work.

[9] Secure Hash Algorithm (SHA)

Chapter 3

Previous work

Classification and identification of code are not new concepts; they have been used since the beginning of reverse engineering. This chapter explores some of the research and tools used in code classification and identification. It focuses on function identification, not the classification of a whole program. Then, the chapter provides a detailed explanation of a tool implementing CFG analysis, *GraphSlick*. This tool is expanded upon in Chapter 4 to be able to do function identification.

3.1 Classification and identification of code

Fast Library Identification and Recognition Technology (FLIRT) is an industry-standard tool for function identification. FLIRT is a module in the analysis tool *IDA Pro* used for function identification [78]. FLIRT uses binary matching to identify functions. A database of FLIRT signatures is included with IDA when it is installed. If a signature in the database matches a function in a program, the function in the program is labeled as the matching function from the database. The signatures are based on the first 32 bytes of a function [79]. The fact that FLIRT uses exact byte matching on a limited part of a function makes it vulnerable to wrongly identifying functions with a similar or identical beginning. It was shown in [80] how "FLIRT aware" malware could use this to its advantage. A method to remove the false positives in FLIRT was shown in Griffin *et al.*[81]. Griffin *et al.* used dynamic length byte sequences, *signatures*, and reference heuristics to identify the relations between components in a program. This was mainly used to detect malware components. Library function identification was also implemented as a way to limit false positives in malware classification, but this was not the main focus of the project.

3.1.1 Graph analysis for identification

An alternative to binary matching is using graph representations of a program to analyze it. As described in Section 2.7, this can be done in multiple ways. Graph analysis has been used to classify whole programs and function identification.

Håland[53] used CFG analysis as a method for malware classification. Malware samples were classified into families based on a CFG analysis of the malware's call graph.

One example of function identification using CFG analysis is *BinSign*[82]. *BinSign* used CFG analysis as an alternative to binary pattern matching for function identification. A CFG was created by using the basic blocks of a function. In each basic block, a fingerprint was created by looking at the number of instances of each instruction in the block, what mnemonic was used in each instruction, and its operand types. *BinSign* focused on complete function matching and tried to match each function with a function from a database of known functions. It showed that CFG analysis was more precise than binary pattern matching at identifying complete functions.

Research has also been done in the identification and classification of inlined functions. This has been done by creating a graph based on the order in which instructions are executed, like using basic blocks. It has also been done by ordering instructions based on how they affect data and if they affect the same data. Qiu *et al.*[83] used a method tracking execution flow graphs. Execution flow graphs are a version of CFGs in which a graph is created based on the context of each instruction. An execution flow graph is created by determining which instructions affect each other when executed. This is done by tracking the data affected by each instruction; if a subsequent instruction is affected by a previous instruction, they are part of the same execution flow. If an instruction overwrites the data affected by a previous instruction, it is part of a different, subsequent execution flow. Qiu *et al.* used execution flow graph to identify library functions in [83] and [84]. Execution flow graph analysis was effective at identifying inlined functions in situations where the execution flow could be predicted. However, it encountered situations where the execution flow could not be determined. In those situations, the method was unable to identify known functions.

3.2 GraphSlick

In 2014, a project called *GraphSlick*[85] was presented. GraphSlick was created as a plugin for IDA Pro in an effort to use CFG analysis as a method to detect duplicate code, developed by Rahbar *et al.* It was released and presented in a plugin creation competition organized by HexRays[1]. The development began in 2013, according to its GitHub repository, and has not been updated since November 2014 [85]. GraphSlick is described as a tool for:

> "automated detection of inlined functions. It highlights similar groups of nodes and allows you to group them, simplifying complex functions." - HexRays[86]

[1] `https://hex-rays.com/contests_details/contest2014`

It analyzes a CFG created from basic blocks to find sequences of blocks that are isomorphic. Isomorphic meaning, "being of identical or similar form, shape, or structure"[87]. If any are found, they are likely inlined functions or code copied to multiple places. GraphSlick is then able to outline the code blocks and show them as subroutines.

The GraphSlick tool is split into two parts. The first part, which is referred to as the *GraphSlick-GUI*, handles visualization and integration with IDA and is written in C/C++. The *GraphSlick-GUI* only provides visualization and a user interface and is, therefore, not relevant to the thesis. The second part is called *bbgroup* and is written in Python. *Bbgroup* is the actual algorithm and does the duplication matching. Bbgroup is not directly a plugin to IDA Pro but uses IDA's Python libraries and has very limited functionality outside an IDA environment. In the thesis, the focus is on bbgroup. The name *GraphSlick* is used to refer to the analysis tool and method in its entirety, whereas the name *bbgroup* is used when referring to the code implementing the analysis method in GraphSlick. GraphSlick does not have any functionality related to function identification, but the analysis method used could be modified to implement this, this is explored in Chapter 4. In the following sections, a detailed explanation of GraphSlick is provided. This is necessary as we had to change the code to make it work for our experiment.

3.2.1 GrahpSlick's analysis component

The analysis component of GraphSlick is called Bbgroup. Bbgroup is the most extensive part of the GraphSlick code, and the code-matching is implemented in bbgroup.

As described earlier, bbgroup is not a direct IDA plugin and could be used in a standalone capacity. However, in the standalone mode, the functionality is limited to only searching graphs that have already been generated from a run in an IDA environment. This means it cannot generate new graphs or perform any new analysis outside the IDA environment. The standalone version is, therefore, almost exclusively used to test and verify that the plugin's components work.

The flow when using the Bbgroup is shown in Figure 3.1. Bbgroup starts by iterating through each basic block in a function and generating two hashes for each basic block called *hash_itype1* and *hash_itype2*. These hashes are further explained in the following paragraphs. Each basic block also gets a reference to its predecessors and successors. The predecessors and successors are the previous and next basic blocks from a control flow perspective, as explained in Section 2.7.2. This is an equivalent structure to IDAs "Graph View". The predecessors and successors are taken from IDAs analysis and decompilation. Bbgroup does not do any of its own analysis to determine the control flow of a function. In Figure 3.2, node A would have nodes B and D as its successors and no predecessors. Nodes B and D have A as their predecessor, B has one predecessor and two successors, and C and D have no successors.

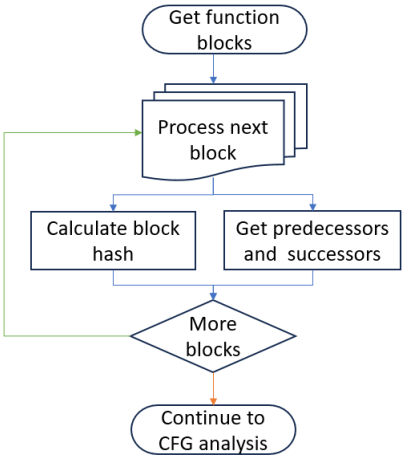


Figure 3.1: Bbgroup functionality flowchart shows the main steps bbgroup does when creating a CFG of a function. It iterates through every block in a function, calculating the hash of a block and getting its successor and predecessors. This is done for every block.

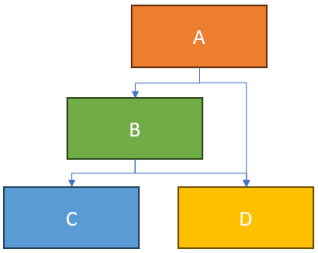


Figure 3.2: Simple graph representation of function with four blocks. Each block has a letter used to reference it and links between blocks that show the possible control flow. These links show which blocks are the successors and predecessors to each other.

When referring to blocks, bbgroup uses the IDAs block ID. A block ID is a number given to each block in a function; the first block in a function is block 0, and each subsequent block is given an incrementing number. The blocks are given a number based on their position in the CFG in the function. A block ID is unique in a function, but across different functions, the same block ID will exist many times. When referring to blocks across functions, the address of the first instruction in the block is used, or the block ID as well as the function name or address.

The type1 hash is a SHA1 hash based on each mnemonic in the basic blocks. This is implemented using an instruction's `itype`-element. The itype is a property of an assembly instruction and contains the "Internal code of instruction" according to the IDA SDK [88]. IDA uses this internal value to keep track of each instruction. However, as shown in Figure 3.3, instructions with the same mnemonic

but different opcodes, get the same itype. The instruction at address *0x00401DA0* starts with the bytes *0x8B* while the instruction at address *0x00401DA7* starts with *0xA3*, however both have itype equal to '122'.

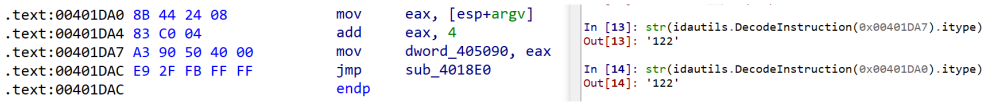


Figure 3.3: The figure shows that itype does not depend on opcode but on mnemonic. The two *mov*-instructions at address *0x00401DA0* and *0x00401DA7* have different opcodes, starting with 8B and A3. The itypes of both instructions are '122', as shown on the right side of the figure. *idautils*, part of the official IDA Pro Python API, is used to show the itype.

The type2 hash is a SHA1 hash using the itype of instruction and the operand types. This method is more robust since it differentiates between the types of operands used. It is, however, not as strict as if the whole machine code was used. The type2 hash would give the same hash value to a block only containing `MOV EAX, EAX` as a block only containing `MOV EBX, EBX`, since it only considers the operand types, not the value. On one hand, this could give unequal blocks of the same hash and create false positives. On the other hand, it is more resistant to cases where the compiler coincidentally uses a different set of registers.

Isomorphic subgraph detection

GraphSlick's analysis component tries to identify duplicate code. This is done by deciding if the CFGs of code parts are isomorphic. If two or more CFGs are equivalent or isomorphic, they are marked as duplicate code. The analysis process GraphSlick uses for isomorphic detection is explained with the help of Figure 3.4. The figure shows a function comprised of ten blocks, named A to J. The function has a single return block, J, and starts at A. The color of the blocks represents its hash value; blocks with the same color have the same hash value.

The analysis would start by identifying two hash-equal blocks. This is done starting from the top of the graph. In Figure 3.4, blocks B and F would be the first pair of blocks to satisfy the criteria. The analysis would then continue using blocks B and F as the potential starting blocks for two duplicate code parts. From B and F, their successors will be checked to see if they are identical. This would be the case for the pair C and H, but not for D and G. As the pair C and H are equal, the process would be repeated for their successors. This would identify that the pair E and I are identical. As J has both C and H as its predecessors, it would be disregarded since a block can not be shared between the two subgraphs. Finally, E and I will be checked; since both only have J as their successor, it would be disregarded as explained above. As no further blocks were identified, the analysis would end, and the two subgraphs comprised of {B, C, E} and {F, H, I} would be marked as two instances of duplicate code.

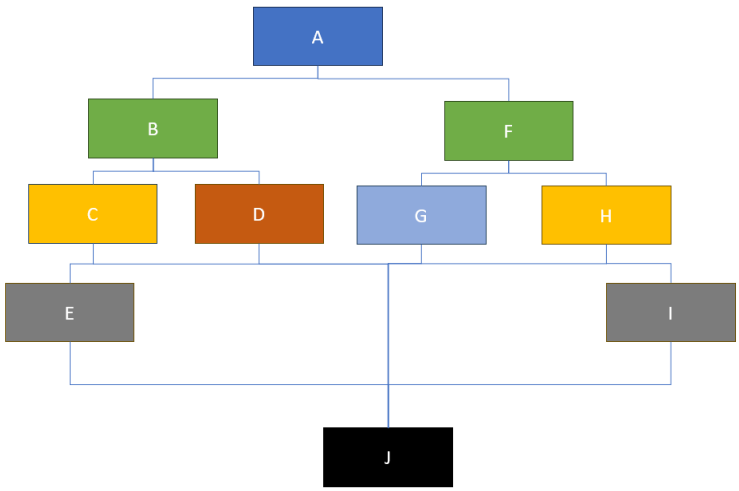


Figure 3.4: The figure shows a graph with two isomorphic subgraphs. The graph of nodes {B, C, E} is isomorphic with nodes {F, H, I}. The color of the node represents its hash value.

3.2.2 GraphSlick as an inlined function detection tool

When looking at code from an inlined function, a function that the compiler inlined as part of an optimization as discussed in Section 2.5, the code itself does not differ from code in the rest of the function. A regular function will typically start with a prologue and end with an epilogue as described by Sikorski and Honig[23]. A prologue is saving the base pointer to the stack and setting up a new stack frame, and the epilogue is setting up a return value, resetting registers, and returning. An inlined function is not a separate function and does not have these markers. Detecting inlined functions based on some specific instructions or actions is, therefore, not possible. To find inlined functions another method is needed.

When a program is created, a lot of code is used more than once throughout the program. This is typically done by putting the code that will be used multiple times in a function. The function could then be called from multiple places. Included libraries are examples of code likely to be used multiple times. If functionality from a library is needed it is probably needed more than once. This is applicable to both standard libraries, like the string library in C/C++ with functions like `strcmp()`[89], and third-party libraries, like the graphics library qt [90]. This assumption is critical to make it possible to use GraphSlick to detect inlined functions.

A graph view of an example function is shown in Figure 3.5. The color of each block represents its hash, and the letter is a way to address a specific block. The function starts with node A, and the control flow then diverges to either B or F. It could then take multiple paths but always ends with node I. The group consisting of nodes {C, D, E} and {F, G, H} would be examples of similar subgraphs. Both groups contain a primary node, C and F, each with one "yellow", {E, H}, and one

"blue", {D, G}, and successor node. The two groups are, however, not examples of potential inlined functions as there exists an extra edge in the graph in one instance and not in the other. The edge from B to E is an edge that comes from an "outside" block, a block not part of the subgraph of the inlined function. It would violate one of the properties defining a function. As explained in Section 2.5.1, a function can only have a single "entry point". This also applies to inlined functions.

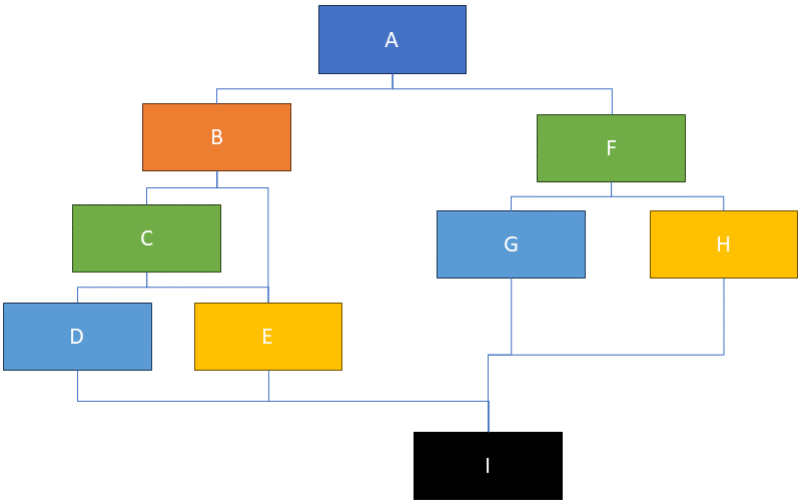


Figure 3.5: The figure shows a graph of a function. The color of each block represents its hash value. The function has two groups of similar nodes, green, blue, and yellow. The two groups are not isomorphic subgraphs as there exists a link between two nodes in one of them, B to E, but no similar link exists in the other group.

Since the hashes are somewhat fuzzy, by only using mnemonics or mnemonics and operand types, it would not be possible to assert with complete certainty that the nodes in the two groups, {C, D, E}, and {F, G, H}, are equivalent. However, the certainty would increase if a larger group of matches were found. Figure 3.6 shows a small section of a larger function. In the figure, two groups of blocks are colored; each node in one group has a "twin node" in the other group with the same hash. The "twin nodes" have the same hash, and predecessors with the same hash, as described in Section 3.2.1. As the two subgraphs in Figure 3.6 contain a significant number of blocks, it would be more likely that they are equivalent compared to isomorphic subgraphs with fewer blocks. The number of blocks a set of subgraphs is comprised of increases the likelihood that they are equivalent. GraphSlick provides the option to define the minimum number of nodes a subgraph needs to be composed of before it classifies graphs as equivalent.

GraphSlick is a plugin mainly designed to find subgraphs that share similar blocks. The similarity of blocks is based on the hash values assigned to each block as described in Section 3.2.1. If we assume that a given program is optimized to

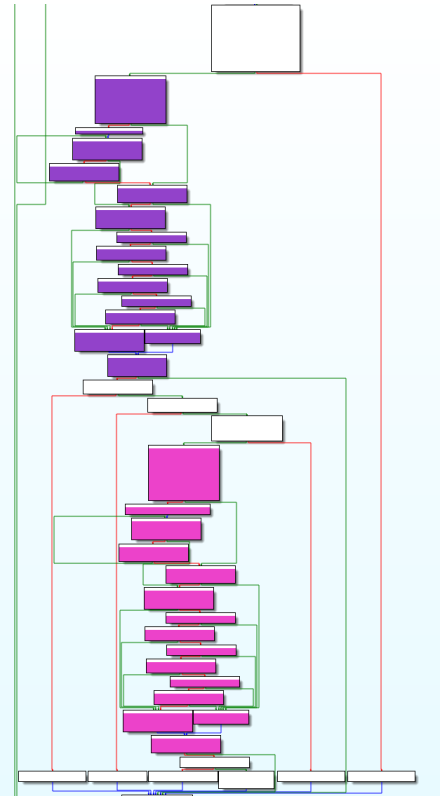
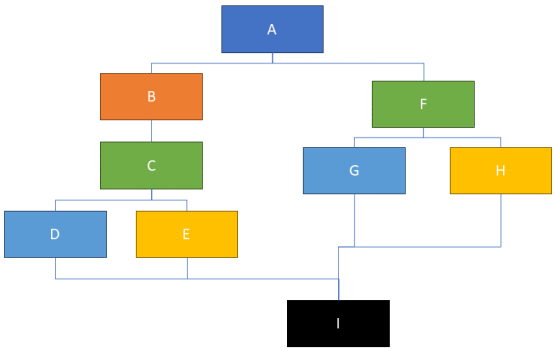


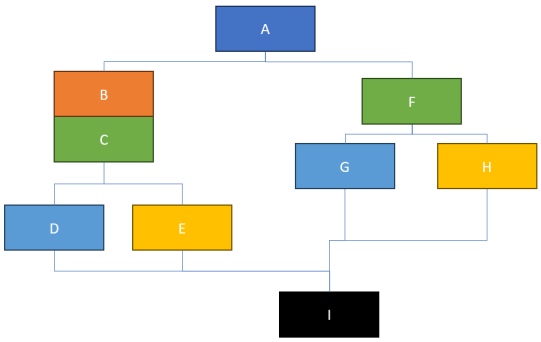
Figure 3.6: Part of a function is shown using IDA's *Graph View*. The portion of the function shows two matching subgraphs. The matching blocks in the matching subgraphs are colored. The purple blocks are part of one subgraph, and the pink blocks are part of the other.

inline functions and functions are called more than once, a set of subgraphs that share similar blocks could be a single function that has been inlined in different places. There is also a chance it could be a similar piece of code manually copied to multiple places.

A function has a single entry point, takes in some data, and returns to where it was called from [91]. The same principles apply to an inlined function since it originally was a function before the compiler optimized it. An inlined function does have some differences in how data is passed to it, how it "returns," and how it is "called," as described in Section 2.5.1. Since functions only can have one entry point, this can be used to detect if a set of similar blocks is an inlined function or not. In Figure 3.5, node B has two successors, C and E. The nodes {C, D, E} are, therefore, not able to be an inlined function since there would have been two entry points, C and E. However, if the function looks like in Figure 3.7a, the two groups, {C, D, E}, and {F, G, H}, could be a single function being inlined twice in the same function.



(a) The figure shows a graph of a function. The color of each block represents its hash value. The function contains two isomorphic subgraphs of blocks {C, D, E} and {F, G, H}.



(b) Same as Figure 3.7a but with a realistic view of how the graph would look, blocks B and C concatenated. Block B and C would not be different blocks in reality, but they are colored differently to make comparison with Figure 3.7a easier.

Figure 3.7: The graphs show a similar function to the one shown in Figure 3.5. The function in Figure 3.5 has a link between block B and E, which is not present in this figure.

Because of how control flow works, instances of inlined functions do not create distinct "entry points" for inlined functions since they are part of the function they are being inlined into. This creates combined blocks that contain part of the main function and part of the beginning of the inlined function. Separating where the inlined function begins and where the main function ends is not trivial. A single block could contain the beginning of the inlined function and parts of the main function. In Figure 3.7a, block B and block C would, in reality, be part of the same block, and the graph would be more like the graph shown in Figure 3.7b. This makes detecting identical subgraphs more difficult. It is easy to see that blocks {F, G, H} are equal to {C, D, E}, but this process is more complex if block C is not a separate block.

Chapter 4

Implementation

In the following chapter, how GraphSlick can be modified to be used as an inlined function detection and identification tool is explained. It also explains problems related to implementing a technique for inlined function detection in compiled programs and possible solutions. Inlined function identification and related issues are also described. A solution for inlined function detection and identification is presented. Testing of the method is described later, in Chapter 6.

We considered GraphSlick, Bindiff, and Diaphora as baseline for the work done in this thesis. Bindiff and Diaphora are industry-standard tools for comparing programs. They are more refined, offer more expansive functionality, and have a more polished user experience compared to GraphSlick. This makes them popular tools for reverse engineering. However, they are more complicated to modify. The extensive functionality means that more components affect the program, and modifications to them must account for this.

GraphSlick is an open-source CFG-based analysis tool integrated into IDA Pro. It is designed more like a research project or proof of concept. It was selected as it is a more specialized tool designed around CFG analysis and has a more narrow focus compared to the other two. It does not offer functionality outside the main CFG analysis component, which makes it easier to understand and modify. Therefore, GraphSlick was seen as the best candidate for testing CFG analysis techniques.

As previously described, GraphSlick was created around ten years ago and does not function in its current state on updated systems. To make it work with the current version of IDA, version 8.3.230608[1]; some updates had to be implemented. The changes were mainly related to switching from Python 2 to Python 3 and the naming convention in the IDA-python API. These changes are documented in Appendix A.1. This was done not to improve GraphSlick but only to make a version close to the original functioning in the thesis. The following sections describe modifications done to GraphSlick in an effort to expand on its functionality.

[1]This version was the most updated version when work on the thesis was started. For consistency and reproducibility, the following version is always used unless otherwise specified: Version 8.3.230608 Windows 10 x64.

4.1 GraphSlick modifications

Some modifications are needed to enable GraphSlick to detect inlined functions. These modifications are inspired by the literature described in Chapter 3. Different methods were tested to overcome the challenges described in Chapter 3 and identify inlined functions. This chapter explains the different methods that were tried. At the end of the chapter, the solution that was settled for is explained, and why this solution was seen as optimal.

4.1.1 Identifying the function start

One way to detect inlined functions is to discover their beginning. As described earlier, an inlined function does not contain the same prologue or signature start as normal functions. Arguments into inlined functions are not easy to detect, since they use the same stack as the main function or they could be passed in registers. Using registers to pass arguments into functions is not unique to inlined functions, however, inlined functions do not have any call instructions referencing their start. Therefore, no calling convention applies, and each instance of an "inlined call" could be different regardless of the calling convention in the rest of the program. Without any function prologue and no defined calling convention, a different method must be used to identify the start of an inlined function.

Another way to detect a function is to use references. Normally, functions have pointers that reference their start. By going through all the references in a program, some of them would reference the start of a function. This would also be the case with inlined functions. However, as seen in Figure 3.7b, an inlined function could start as part of another block. It might have been possible to assume that some inlined functions were more like blocks {F, G, H} in Figure 3.7b. Since block F is a standalone block it would have references pointing to it. This method could, therefore, in some cases, have been used to detect the beginning of inlined functions. The problem is that this method is unreliable, and functions without references to their start would not be detected.

4.1.2 Library lookup

If some attributes of the program are known at the time of analysis, this can be used in inline function detection. Some of the inlined functions in the program being analyzed could be from external libraries that are statically linked. Assuming the external libraries are known, this could be used to detect functions from that library in the currently analyzed executable.

A possible way to implement this would be to create a signature for each of the functions exported by the library. Then, each of these signatures could be searched for in the program being analyzed. If a signature is found, this will both detect the location of an inlined function and identify the function being inlined as the library function. The signature for a library function could be created as CFG for the function using GraphSlick. A compiled version of the library would

need to be generated to match CFGs from the library with code in the executable. This compiled library should preferably be compiled with the same settings as the executable; this will make it more likely that the signatures in the library match parts of the code in the executable.

This method would likely be quite effective in detecting statically linked functions in an executable. It might, however, have some problems when trying to identify inlined functions. Inlined functions do not have a clear function prolog, or an easy way to identify the location where the original function ends and inlined function starts as discussed in Section 3.2.2. This lack of a clear start would make matching an inlined function with a library signature challenging.

4.1.3 Duplicate code detection

Inlined functions are likely not inlined only once, and multiple instances of the same function inlined in several places could exist. The different instances would probably not be identical as described in Section 3.2.2. They would, however, still share most of the same code. It would, therefore, be possible to detect different instances of a known inlined function. Or it would be possible to assume that some identical pieces of identical code are part of an inlined function.

By detecting duplicate code parts, it is possible to assume that some of these duplicate code pieces are part of an inlined function. If a part of an inlined function is known, it would be possible to expand from the known part and check if earlier or later parts are part of the same function. By doing this search in a loop or recursively, it would be likely that the entire inlined function is found. This method depends on a technique to identify if a piece of code is part of the same inlined function. A technique to solve this is described in Section 4.2.

Like the previous ones, this solution has some weaknesses, and some assumptions are needed. The most obvious weakness is the assumption that a function is inlined more than once. This assumption is likely not always true and may lead to cases where a function is only used once and, therefore, not detected. Another weakness or problem to overcome is differentiating between inlined code from libraries and other duplicate code pieces found in the program. Most of the duplicate code will likely not be part of an inlined function, a type of filtering or sorting is needed to decide if a duplicate piece of code is an inlined function or something else. An inlined function would also need to be matched with a library to identify it.

The *Duplicate code detection* solution was seen as the one with the most potential, this method is used as the basis for further testing. In Section 7.3 this choice is further discussed.

4.2 Splitting of basic blocks for duplicate code detection

GraphSlick, as developed in 2014, is not able to detect inlined functions as described in Section 3.2.2 and shown in Figure 3.7b. This is mainly because inlined functions share their beginning basic block with code from the function they are inlined into. Different instances of the same function inlined in different places would then not have the same hash on the first block. This is shown in Figure 4.1a. In the figure, one function is inlined twice. The inlined function starts as part of block pair A and B and ends with the block pair C and D. Instead of identifying the complete inlined function, GraphSlick recognizes a series of blocks as equal and tries to create subgraphs of equal blocks based on this. This creates several smaller subgraphs since the common starting blocks, A and B are not recognized as equal, and the lack of a common beginning propagates downwards. This can be seen in Figure 4.1a by looking at the green and the brown sets.

A possible solution proposed in this thesis is splitting basic blocks. This method is a three-step process: first, do a normal analysis with GrapSlick, then identify basic blocks that could potentially be the starting block in a graph but are not currently part of one, and finally, split those basic blocks in two and redo the analysis to test whether splitting the blocks improved the analysis. The result of this process can be seen in Figure 4.1b.

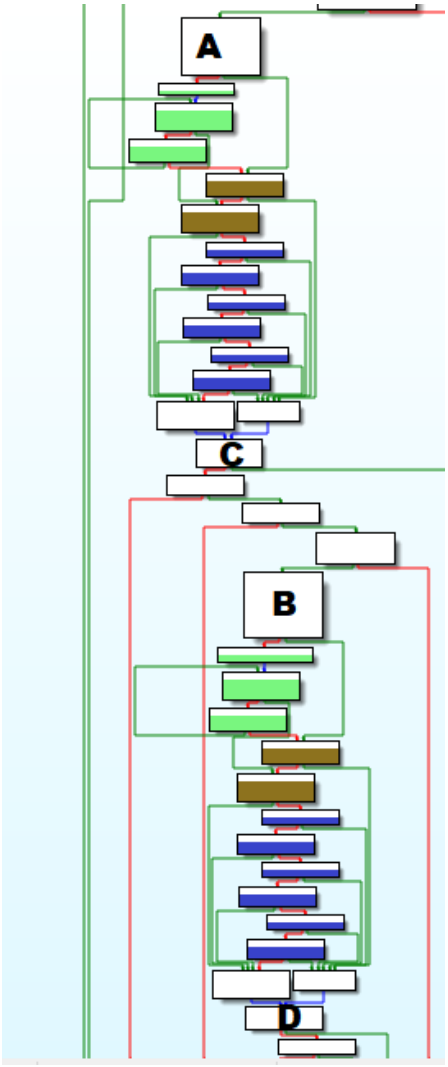
4.2.1 Splitting basic blocks

Splitting basic blocks is a possible solution for finding the beginning of an inlined function. By splitting a single basic block into multiple parts, the CFG analysis can get a more nuanced and detailed picture of the analyzed program. This extra detail negates the problem of having basic blocks containing part of an original function and part of an inlined function.

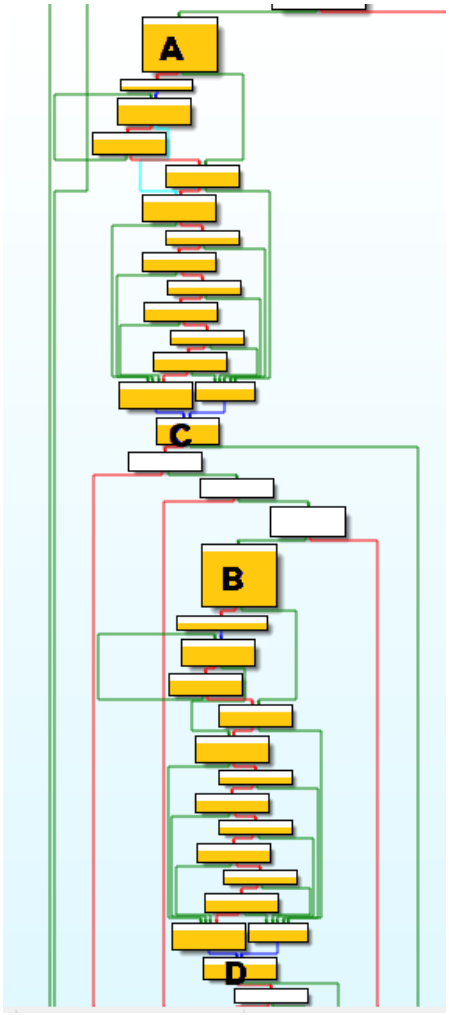
IDA does not naturally have a way to split a basic block into multiple parts. To achieve this, a "virtual" basic block is created. This virtual basic block is not added to IDA's internal program structure and, therefore, does not affect IDA's own analysis process. This virtual basic block is only a part of the CFG used by GraphSlick.

When GraphSlick does its analysis step as described in Section 3.2.1, it begins by creating a CFG of a function. This CFG is created by hashing every basic block and building a graph based on every block's predecessor and successor. This process uses the IDA-API to get a list of all basic blocks in a function. By modifying GraphSlick, it is possible to create modifications to the basic blocks and add custom ones, as shown in Figure 4.2. This process takes a basic block, block A in the figure, and finds somewhere to split it, calculates the hash of the two new blocks, and finally creates links or references to make the new blocks a part of the graph.

Figure 4.2 shows a representation of the internal state of the data in a modified version of GraphSlick during an analysis where a basic block is split. In the figure, the color of each block represents its hash value during the different parts of its



(a) The figure shows the isomorphic subgraphs in a function. The colors of each set of nodes represent their equal counterparts. For example, the two sets of green blocks, right below A and B, are equal.



(b) The figure shows the isomorphic subgraphs after blocks A and B are split. This identifies all the subgraphs in Figure 4.1a as the same subgraph and expands the graphs down to the last blocks, C and D. The entirety of blocks A and B are colored for visualization purposes. However, only the lower split parts match.

Figure 4.1: A graph view of a function in IDA shows a part of a function containing two instances of an inlined function starting in blocks A and C. The blocks identified as isomorphic subgraphs are colored. Figure 4.1a visualises the default analysis and Figure 4.1b visualises the results after splitting two basic blocks.

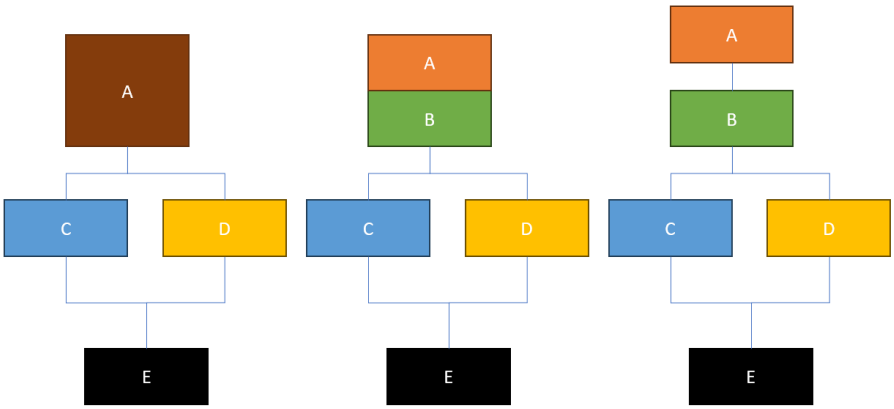


Figure 4.2: The process of splitting a basic block is shown in the figure. Block A is first split into two blocks, the orange and the green. The orange block would start at the same location as the original brown block and, therefore, still has the reference A. Lastly, blocks A and B are separated to show that they are no longer the same block.

analysis. The letter represents the starting address of each block and the lines how they are connected. The process starts with identifying a block to split and where to split it; in the example, it is decided to be split at "address B." Where to split a block is explained in the following section, "*Identifying inlined function starting point*". Then, the hash value of the set of instructions from the start of A to B is calculated. This hash value is set as the new hash for the A block, and the end of the address of the A block is set to the address B. The hash of the set of instructions from B to the end of the basic block[2] is also calculated. Next, a new basic block data structure is created with the start address of B, the hash value of the block is set to the hash previously calculated, the end of the block is set to the same as the original end of block A, and the same successor blocks as the A block originally had. Finally, A's successor blocks are changed from blocks C and D, and to B. The result of this process is a graph compatible with the steps done in GraphSlick's analysis process.

The splitting of a basic block is a computationally expensive task. It requires the recalculation of multiple hashes, which in and of itself is a costly operation. It also modifies the graph GraphSlick uses for its analysis, so any analysis already done by GraphSlick must be redone. This is further discussed in Section 7.3.3.

[2]The end of the basic block is normally the start address of one of the succeeding basic blocks, C or D in the figure, but not necessarily.

Identifying inlined function starting point

Splitting blocks is computationally expensive, and it would be unfeasible to try splitting every basic block. Therefore, deciding which blocks to split is an important part of the process. As previously discussed, an inlined function will likely be optimized and fitted to the function it is inlined into. This primarily affects the beginning and end of the inlined function. The CFG analysis used in this thesis uses a single starting node and tries to find isomorphic subgraphs based on successor nodes to a starting node. Identifying the correct starting node is a requirement for the method to function as intended. Without a common starting node, the analysis cannot connect different branches of a function. This was previously shown in Figure 4.1.

In Figure 4.1a, GraphSlick could not connect the different parts and instead created multiple smaller subgraphs. However, blocks following where the inlined function started were detected as a matching set of blocks. As Figure 4.1b shows, splitting the block pair directly preceding the first detected isomorphic subgraph, enables GraphSlick to detect the entire inlined function as a single subgraph instead of multiple smaller ones. This indicates that it would be possible to use the analysis done by GraphSlick as an initial estimate of which block would be sensible to split. Trying to split the block directly preceding the first set of detected isomorphic graphs is likely the start of an inlined function. All blocks directly before all instances of the same isomorphic graph must be split and split in the same location for an analysis to be effective.

A block containing the beginning of an inlined function and part of the original function, would have the inlined function at the bottom. It would, therefore, be possible to detect the address where an inlined function starts. By comparing from the last instruction in a block and continuing upward, it would be possible to estimate where the inlined function starts. The exact instructions would not be possible to determine. There could be differences in how different instances of an inlined function are optimized at its beginning. There could also be instances where previous instructions before an inlined function are similar, which would make the technique determine that a part of the original function was part of the inlined function. Determining exactly where an inlined function begins is not necessary for function identification, as is described in the next section.

4.3 Identification of known functions

This section explains how an inlined function can be identified as a specific function from a known program. The method described in this section assumes that some code in a function is detected as a potentially inlined function, i.e., a set of connected basic blocks with a single "entry point" and no external references into it, as described in Section 2.5.1 and shown in Figure 4.1.

After detecting a potential inlined function, identifying which function is inlined is the next step. Identifying unknown programs and understanding them

is the main task in reverse engineering. Identifying code can be done on several levels. Methods used in malware classification analyze a complete program on a higher level and try to categorize it into malware families or groups. Function identification techniques, like FLIRT, analyze a program's individual components on a lower level. CFG analysis, when the CFG is based on basic blocks, also analyzes a program on a lower level, like FLIRT. CFG analysis as a method can also be used in classification and grouping if it's done on a higher level.

The identification of something unknown depends on the existence of something already known that the unknown can be matched with. If the unknown matches the known, the unknown is identified as the known. In function identification, this means that a function is identified if it matches a known function. FLIRT uses a dataset containing a binary mask of known functions. This is used to compare functions detected by IDA and check if they match a function in the dataset. If a match is found, IDA labels the matching unknown function as the matching known function and adds the correct arguments.

FLIRT can not be used directly to identify inlined functions. If a function with an FLIRT signature is inlined, the FLIRT signature will not match the inlined version of the function. The inlined function does not have a precise and easily identifiable starting location. The FLIRT signature depends on the start of a function to be able to find it. Even if the start of an inlined function were accurately identified, FLIRT would still be unable to detect it. FLIRT relies on looking at the beginning of a function to identify the arguments passed to the function. Inlined functions do not have a detectable calling convention, and the "calling convention" used to "call" an inlined function could differ across a program. Outlining an inlined function could be a possible way to make FLIRT able to identify inlined functions, but this is likely unfeasible. To outline an inlined function, the entire inlined function would need to be detected, and the arguments and variables used in the inlined function would need to be identified. During the optimization process of a function containing an inlined function, a compiler could make the inlined function access variables belonging to the function it is inlined into. The arguments needed by the inlined function would also need to be identified. Outlining an inlined function includes setting up a stack and accessing arguments passed to the function. If an analysis tool is able to separate out an inlined function, it already has enough information about the inlined function to identify it.

Instead of using FLIRT, this thesis proposes using CFG analysis to identify an unknown inlined function. As described in Section 3.2.2, CFG analysis can be used to detect similar/duplicate code. An inlined function should be quite similar to a non-inlined version of the same function. A similar technique to the one used for duplicate code detection could be used to identify an inlined function. By comparing the inlined version to a non-inlined version of a function it would be possible to check if they are similar.

4.3.1 Function matching

An inlined and a non-inlined function will not be identical even if they are compiled from the same function. From a CFG perspective, the graph structure will likely be similar, but the content of individual blocks may differ, and parts of the graph may be missing.

The first block in a function will not be equal to an inlined function. This makes using the exact same analysis method described in Section 3.2.2 difficult. That analysis method depends on the existence of a 'starting block' to which all subsequent blocks are related. Since the first blocks, the 'starting blocks' for the analysis, differ, this method would not be able to traverse through the graph and test if subsequent blocks match. Given a CFG of the inlined function and the non-inlined function, replacing the first block in the two CFGs with a fake, predetermined, equal block would be possible. This would make the first blocks match and could make the duplicate code detection technique work, but it would rely on the function start, the difference between the inlined and non-inlined function, being contained in a single block. If a function has several blocks used to initialize it, meaning more than one 'starting block', its inlined counterpart might differ in more than one block. Creating fake starting blocks could limit the function identification to only working on functions with one starting block.

The CFG comparison could also be done by ignoring the lack of a common starting block. This would make the comparison method similar to GraphSlick without block splitting. The advantage of this is that it would work without identifying common starting blocks. The disadvantage is that some blocks would not find a match, and holes in the graph could appear. As shown in Figure 4.1, multiple split subgraphs could create situations where a block is lacking a group.

Traversing the blocks in the two CFGs and finding matches without needing them to be connected to a preceding block would solve the 'starting block' problem. However, this would make the matching method more prone to wrong identification in cases where blocks are equal but the structure of the CFGs differs. By only matching block hashes, the method would not be able to differentiate between graphs with the same blocks if the blocks swapped positions.

The differences between an inlined and non-inlined function is tested in Chapter 6.

4.3.2 Partial matches

Regardless of the method used for matching an inlined function, the whole function is unlikely to match its non-inlined counterpart. Both the 'starting block' and compiler optimizations make this unlikely. Therefore, a score needs to be given for each potential identification. The identification with the highest score is chosen as the most likely known function the unknown function equals.

The score is given based on the percentage of blocks matching between the known and unknown functions. The 'starting block' will be deducted from the

number of blocks as this block will not match, and any blocks containing a return-instruction[3] is also ignored when calculating the number of matching blocks.

4.3.3 Known function dataset

The dataset of known functions needs to contain a large number of library functions to be able to match a large percentage of unknown functions.

The dataset would also need to include several instances of the same function. Some high-level language functions can be called with multiple different arguments. One example is the C++ function `std::string::compare()`. This function refers to different machine code instructions depending on the arguments passed to the function. If a programmer calls the string compare function like `std::string::compare(std::string*)` in machine code, this will not be the same function as if `std::string::compare(const char*)` is called. Since these are two different functions from a machine code perspective, the dataset of known functions should optimally include both versions. The difference between the functions might not be large in some cases, or the difference is so small that the instructions/mnemonics used are identical. In that case, the analysis process would not be able to differentiate between the known functions.

This thesis does not focus on the known function dataset. The main focus is the detection and identification process; generating an extensive list of known functions is not necessary for this process but will be necessary if the work done in this thesis is to be used in real-world scenarios.

[3]This only applies to the known function, as the unknown is an inlined function and inlined functions does not have a return instruction in them.

Chapter 5

Verification and testing

The following chapter describes a framework for verifying inlined function detection and identification. The framework is designed to test arbitrary analysis methods, compare analysis tools, and validate techniques. Running a large-scale test is beyond the scope of this thesis, so the entire framework has not been tested. This requires more time than is available in a master's thesis and is left for future work. In Section 5.1, the test framework is described. The main components of the framework and the goal and challenges of each module are explained. Section 5.2 explains how each module can be implemented and how they interact.

5.1 Test framework

To verify any analysis process on a large scale, a test model has to be designed. Different levels of software testing exist, like unit tests and integration tests. Different levels of testing are used depending on what the test is supposed to uncover. Unit tests are designed to test smaller components or functions in a program, while integration tests are used to test a larger system [92], [93]. The test framework described in this section is designed to test an entire analysis method or tool.

Any test framework is composed of multiple components that need to work independently of each other. To accomplish accurate results, each component must be tested to ensure correct functionality. This ensures that each component works as intended and that the integration with each component is correct. However, the main focus of this thesis is on creating an analysis technique, and testing of the test system is not discussed further.

5.1.1 Test framework design

The system described in this section has been chosen as a suitable design for testing function detection and identification tools. The system is module-based to accommodate multiple test samples. Each module takes data in a specified format as input and returns it in the format expected by the next module. Any test system needs to take input data, process it, output results, and verify those results.

A function detection and identification test framework needs an input program, analyze the program, and verify the analysis. To have more control over the input data, a pre-compiled program is not used as test data. Instead, source code is taken as input and compiled into a program. Based on this, the test framework needs to be comprised of the following steps.

1. Generate test data by compiling a source code program
2. Analyze the test program
3. Generate result files based on the analysis
4. Verify the results by comparing them with data from the source code and the compiler
5. Save the results
6. Go back to Item 1 and compile a new program or the same program with new compiler flags

This list of steps can be visualized in a figure. Figure 5.1 shows the main components of the model. The model is based on the following modules: the compiler-/linker, the analysis module, and the verification. Each module can be replaced independently of the others as long as the output from each module is in the expected format. Test data generation, ie. compiling source code, is a step that would likely need to be changed depending on what program is compiled. This step could also be modified depending on the compiler that generated the test samples. Each module is further explained in Section 5.2.

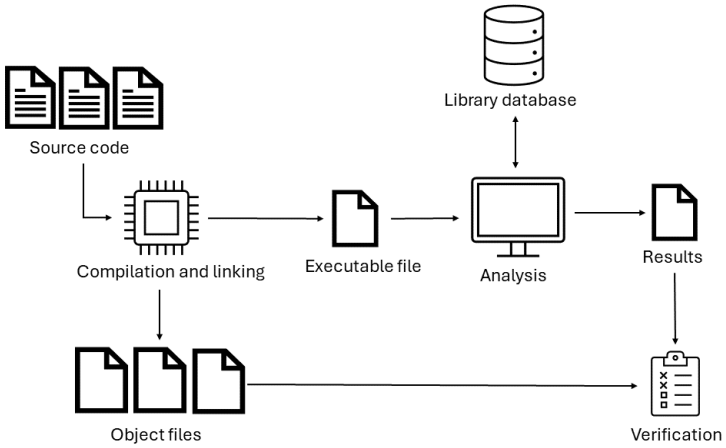


Figure 5.1: The figure shows how a test system for program analysis can be designed. It expands upon the figure showing how a program is created from source code, Figure 2.1, with the components needed for program analysis. It replaces the program execution with an analysis step. A database containing library functions and a verification module is also added.

5.1.2 Test components

The test framework needs to be able to take in test data, source code, analyze it to find inlined functions, identify whether the inlined functions are from a library and from which library, mark the found functions, and verify that the results are correct. The following paragraphs explain briefly the modules from Figure 5.1.

Source code, Compilation and linking & Executable file

The test data module of the test framework is responsible for handling test data and passing it to the analysis module. The module needs to create a program that can be analyzed by the analysis module. To test if inlined functions detection and identification, the test data need to be compiled into a computer program. Most real-world machine-code programs are in the form of a PE file on Windows[1].

Analysis & Library database

The analysis module needs to integrate an analysis tool or technique, like the analysis technique described in Section 4.1. This includes taking in an executable file, finding the inlined functions, and looking up in a "library database" if any of the inlined functions are known. The tool, however, needs to return its results in a "result-object" instead of displaying them to the user. This result object needs to contain the inlined functions that have been found, the location of said functions, and what function they are if that information has been found. Further description of how the analysis process must be modified to the test framework is described in Section 5.2.2.

Verification

The verification step is the step that takes in the results and determines if the analysis step identified the correct inlined functions. The results could include any combination of true positives, false positives, true negatives, and false negatives. True positives would be inlined functions that have been detected and correctly identified. False positives would be parts of the code that are identified as an inlined function without being one. False negatives are inlined functions that are not identified. True negatives are regular code that the analysis correctly detects as not inlined.

There are different ways to classify false positives and false negatives since detecting and identifying inlined code are two different processes. It would be possible to be more liberal and not classify a case where the analysis tool correctly detects a piece of code as duplicate but fails to identify it as inlined as a false negative. This thesis' main focus is on inlined function identification and not duplicate code detection, and the verification process should reflect this. Therefore, more stringent verification is used.

[1] A ELF file on Linux. However, this thesis focuses on Windows, and PE files are used.

The verification step needs a way to check if the results are correct. The verification process does not know if the results are correct and needs to compare the results with a set of verification data. The verification data needs to be generated with the test data. The implementation of the verification component is described further in Section 5.2.1.

5.2 Test framework modules

This section explains the three main components shown in Figure 5.1, the compiler and linker, the analysis module, and the verification process. The three components are described oppositely compared to how they are used, *Verification* first and *The compiler and linker* last, since how the first steps are done depends on the last step.

5.2.1 Verification module

The verification step in the model needs to take in the results from the analysis tool and determine if the hits are correct. Two metrics need to be tracked: the number of hits and the authenticity of those hits. To be able to track these metrics, information containing which functions exist in the program, where they are located, and if they are inlined need to be gathered. This information could be gathered from the source code or the compiler output.

Source code

The source code contains information about which functions are called from where, the number of times a function is called, and keywords like `__inline`. Extracting keywords like `__inline` or `__forceinline` could, at first glance, be helpful in identifying if a function has been inlined. However, as discussed in Section 2.2.2, these keywords are more of suggestions or guidance to the compiler and are not necessarily followed. The compiler only uses the keywords to add incentives to inline a function. Tracking these keywords in the source code does not provide information accurate enough for the verification process.

Tracking function calls from the source code is also challenging. The compiler does not convert the source code directly into machine code. The compiler is able to modify and optimize a program as part of the compilation process. A part of the optimization process is removing dead code as discussed in Section 2.5. When the compiler detects dead code, it could be removed as an optimization step. If source code were used to create verification data, dead code removal done by the compiler would make the verification data incorrect.

The compiler does multiple sets of optimization runs, and if LTCG is enabled, the linker could do optimization as well. This further creates differences between the source code and the generated machine code.

Object files

One of the final steps during the compilation process is the generation of object files. Object files contain machine code, like the final program, but the machine code is not linked together and is instead spread across multiple files. Therefore, these object files could be used to detect if a function has been inlined. Since object files are created in one of the final code generation steps, they do not significantly differ from the final program. One difference between object files and the final program is that object files contain symbols. The name of a function is one type of symbol.

The linker compiles and produces its own object files if LTCG is enabled. In the Microsoft C/C++ compiler toolchain, these files have the `.iobj` file-extension instead of the `.obj` file-extension. Object files created during the linker stage do not differ significantly in structure or function from "normal" object files. LTCG does, therefore, not significantly affect how object files are used.

By combining object files and source code, it would be possible to identify which functions were inlined. If the source code of a program contains two functions, A and B, and function B is called from function A, the object files could be analyzed to identify if function B was inlined. If the object files do not reference function B, it would indicate that function B was inlined or removed by one of the optimization steps. To decide if it was inlined or removed because of optimization, the same program could be recompiled with function inlining disabled[2]. If the object files now contain a reference to function B, the function was inlined. Since the final program lacks symbols, identifying which functions were inlined would be difficult. This is not a problem with object files. Consequently, object files could be used to create a list containing which functions were inlined, approximately where they are, and how many instances of each inlined function exist. This list can be compared with the result from the analysis step to verify if the analysis was correct.

5.2.2 Analysis module

The analysis module needs to integrate the analysis tool into the test system. The tool implementing the method described in Section 4.1 would need to automatically analyze the program and output the results in a format understood by the verification process. As described above, the verification process needs to know which functions were identified and where they were detected. The analysis tool first detects potential inlined functions and the address of that inlined function. The detection is based on duplicate code detection. The number of instances of each detected inlined function is the same as the number of duplicate instances of that code.

The verification process would not be able to assert with certainty which address an inlined function is located at. Object files are generated before the linking

[2]The *'/Ob0'* compiler flag in Microsoft C/C++ compilers disable function inline expansion [40]

step, where addresses are assigned. However, the object files do contain information about which function an inlined function is inlined into. Tools like IDA Pro are able to take an address and identify which function that address is contained in. The analysis tool can use IDA's interface to convert the address of an inlined function into a function reference.

The analysis tool must output the following for each inlined function: its location (which function it is inlined into), the group it is part of, and which known function is associated with that group. The analysis process would be able to verify whether this information is correct based on information from the object files.

Compiler metadata

Information about the compiler would be needed to ensure that the correct version of a known function is chosen when trying to identify an unknown function. As described in Section 2.2.4, compiler detection is possible. Reverse engineering tools can identify several different compilers and new techniques and methods for compiler identification have been presented in research papers [94]. Some compilers also add compiler information, making compiler detection and rough identification easy, an example is the RICH header, described Section 2.2.4.

Compiler identification is not done as part of the test framework or analysis tool. This is done to limit the scope of the testing and the tools presented in this thesis. Instead, the analysis tool is given the correct information about the compiler. Compiler detection is separate from the scope of this thesis. Still, it is necessary to discuss and address this problem in order to assess the real-world viability of the methods presented in this thesis. This also ensures that any errors in the analysis process and library detection are due to analysis errors, not compiler detection and identification errors.

Known functions dataset

The analysis tool requires a dataset containing signatures for known functions that can be used to identify unknown inlined functions. The dataset must include the name of a known function, the compiler and compiler flags used to create that instance of the known function, and a CFG of its basic blocks. The analysis tool would be able to select which entries in the dataset, based on the compiler and compiler flags, an unknown function is compared to.

A part of the analysis tool would need to generate the dataset before the actual analysis is done. The ability to create a CFG based on basic blocks is part of the analysis tool, and it would be preferred to continue using this tool instead of developing a completely separate one. A program or library containing the known functions would need to be passed to the analysis tool. The analysis tool would iterate over all known functions in the program, create a CFG of the functions, and extract the names of the functions.

5.2.3 Test code generation

As described in the previous sections, Sections 5.2.1 and 5.2.2, both the verification and the analysis steps depend on the code generation step. The verification needs both object files with and without inlining. The analysis step needs a compiled program with inlining. The test would optimally be done using programs compiled by several different compilers. A selection of compilers was chosen to limit the amount of testing needed and the amount of test data required to be generated. The compilers are selected based on which are most commonly used in the real world. *GCC* is the most common cross-platform C/C++ compilers, while the *Microsoft C/C++ compilers* are commonly used when a program is strictly developed for a Windows platform [95]. The Microsoft C/C++ code generation tools include *msbuild* and *cl*. *Msbuild* is the default tool used to build Visual Studio projects, while *cl* is a compiler used in command line building.

To create a robust testing system, the test framework must analyze and verify multiple pieces of test code. Both differences in the type of program that is analyzed and how the same program is generated could create nuances that affect the analysis process. Generating the test program from source code is most efficient for testing different programs and different versions or instances of a program. By compiling the test data, it is possible to achieve greater control over what is being tested. It is also easier to generate multiple test samples from the same source code by varying compiler settings and compiling the same program with different compilers. Verifying the analysis results is also easier if the test samples are compiled from known source code. There are, however, some disadvantages to compiling the test samples as part of the testing process. Setting up an environment to compile different samples with the necessary dependencies takes time. This process will also need to be repeated for every sample and compiler with which the sample will be compiled. Another disadvantage is the compilation environment; if already compiled samples were used, it might be more representative of the compiler environment of many distributed programs.

Open-source programs are better suited for the testing process compared to closed-source programs. The main reasons for this are availability and reproducibility. The source code for closed-source programs is unavailable to people outside the distributors. Another reason for using open-source is coding standards and quality. Test programs that follow established best-practice coding styles are selected to get the most accurate representation of real-life programs. Large open-source projects have many contributors and are more likely to follow best-practice coding styles. If this is not the case and a single style is not enforced, the different styles of all the contributors will make the code difficult to work with. A closed-source project does not necessarily adhere to the same coding practices, and because the source code is inaccessible, it is impossible to check the code quality.

The test code generation step would also need to generate programs for the known function dataset used by the analysis process. This dataset must contain known functions inlined in the test program. It would need to be compiled with

the same compiler and compiler settings as the inlined functions. As previously described, this will ensure that any correct or false results are because of the analysis process. Separate tests would need to be done to assess the analysis process's ability to differentiate between known functions from different compilers. The known functions can be generated by compiling a library where the known functions are exported or by compiling a program with inlining disabled. This would allow the analysis tool to analyze and extract the CFG of functions, which could be used to identify unknown inlined functions. Whether using a compiled library or a non-inlined version of a program is most efficient depends on the known functions and if a library containing the functions can be compiled as a standalone version. These files must be compiled with symbols to make each known function identifiable.

Chapter 6

Results

This chapter describes the tests on the detection and identification of inlined functions. First, the tools used and their versions are described in Section 6.1. Then, tests related to inlined function detection and the associated results are explained. Finally, the identification of inlined functions is tested. The tests are listed below:

1. Inlined function detection
    a. Test 1: Detection of inlined string comparison
    b. Test 2: Detection of inlined function using compiler keywords
2. Inlined function identification
    a. Test 3: Identification of keyword-inlined function
    b. Test 4: Identification of string comparison function

The first two tests are detecting inlined functions and are described in Sections 6.2.1 and 6.2.2. They are conducted to compare the method's effectiveness for inlined function detection compared to GraphSlick. This is done with an inlined function from a library and an inlined function from the same source code.

The last tests, Sections 6.3.1 and 6.3.2, are for inlined function identification. A signature is created of the functions from the inlined function detection tests, and the inlined functions are identified based on the signature.

In addition, the computational performance of inlined function detection was measured. This is described in Section 6.2.3. It was tested to validate if the method could be applicable as a supplement in a reverse engineering workflow.

6.1 Test setup

The tests were conducted on a Windows operating system using a set of tools. These tools, which are briefly described below, were used in their respective versions as shown in Table 6.1. All programs were compiled to intel-x64 architecture unless otherwise specified.

Windows is the most common operating system used worldwide, as discussed in Section 2.3. Therefore, Windows was used during the testing process for this thesis. The choice of OS does most likely not affect the results much since it is a program running in user space that is compiled and analyzed and not a kernel program. However, some differences might occur in the runtime and low-level function implementation. These differences are further discussed in Section 7.4.

Table 6.1: Versions of tools and Operating System version used during the testing process

Operating System information	
OS Name:	Microsoft Windows 10 Home x64
OS Version:	10.0.19045 N/A Build 19045
Tools and programs	
cl Verison:	Microsoft (R) CC++ Optimizing Compiler Version 19.29.30152
GCC Version:	g++ (MinGW.org GCC Build-2) 9.2.0
Msbuild:	Microsoft Visual Studio Community 2019 Version 16.11.31 VisualStudio.16.Release/16.11.31+34114.132 Microsoft .NET Framework Version 4.8.09037
IDAPro Version:	Version 8.3.230608 Windows x64 (64-bit address size)

The compilers were chosen based on their commonality. Both cl (*Microsoft (R) C/C++ Optimizing Compiler Version*) and msbuild are build programs related to *Microsoft Visual Studio*, and GCC is a common compiler used both on Windows and Linux. Appendices B and C contain the code used in the tests and the result details. The main test parameters and results are included in this chapter.

6.2 Inlined function detection

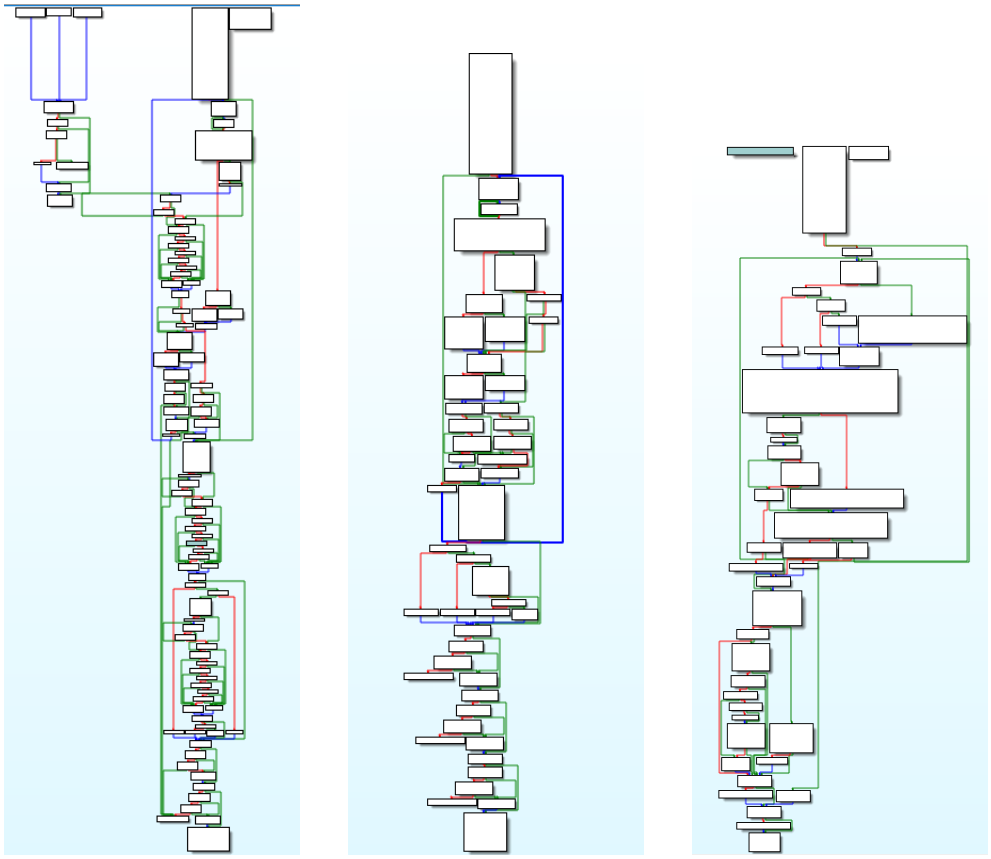
Tests related to inlined function detection are described in this section. First, inlining is done because of compiler flags, and next, due to compiler keywords. Lastly, the performance or time taken by the analysis process is described.

6.2.1 Test 1: Detection of inlined string comparison

The first test, which served to evaluate the detection of an inlined function in a standard library using different compilers, was conducted by compiling the code in Appendix B.1 with the compilers *cl*, *msbuild*, and *GCC*. The compilation was done on Windows as described in Section 6.1. This code included one function with three instances of the C++ function used for string comparison in the standard string library. This is a function part of the standard string library in C++ called

'std::string::compare' [96] . The compilers had optimization turned on and focused on speed. The compiler options can be seen in Appendix B.1.

The compilers produced the following code shown in Figure 6.1. As seen in the figure, the graph view of the same program compiled with different compilers drastically changes the CFG of the program. The "free-flowing blocks" at the top of Figure 6.1a and Figure 6.1c are related to C++ cleanup and *atexit* [97] , and not directly part of the analyzed function. The structure and size of the programs are quite dissimilar.



(a) Compiled by cl **(b)** Compiled by msbuild **(c)** Compiled by GCC

Figure 6.1: Graph view of the different compiled versions of the code in Appendix B.1. The figure shows how different compilers create quite different structures even though the same source code is compiled. The amount of blocks in each compilation is also distinct; the version compiled by cl Figure 6.1a has a lot more blocks than the two other versions.

The programs were analyzed using the original GraphSlick analysis method and the method proposed in this thesis. This method was described in Section 4.2.1. The analysis done using the original method is referred to as *Original GraphSlick*, and the new method, described in Section 4.2.1, is referred to as *Modified GraphSlick*. The analysis process was timed using the Python module

timeit[98]. The time analysis was done using the average of 50 runs. Each run redid the setup and generation of the CFG. This ensured that each run was independent and that previous analyses did not affect the reanalysis of the current run. By redoing the CFG generation each run, some extra time was added. This is not directly part of the analysis, but creating the CFG is a necessity and, therefore, not subtracted from the time.

Table 6.2 shows the execution time for the different analysis runs. Each analysis was done without other analysis jobs running and no other resource-intensive programs running on the system. All timed analyses were done on the same system. The analysis time was measured over 50 runs. The average analysis time and the standard deviation are given in the table.

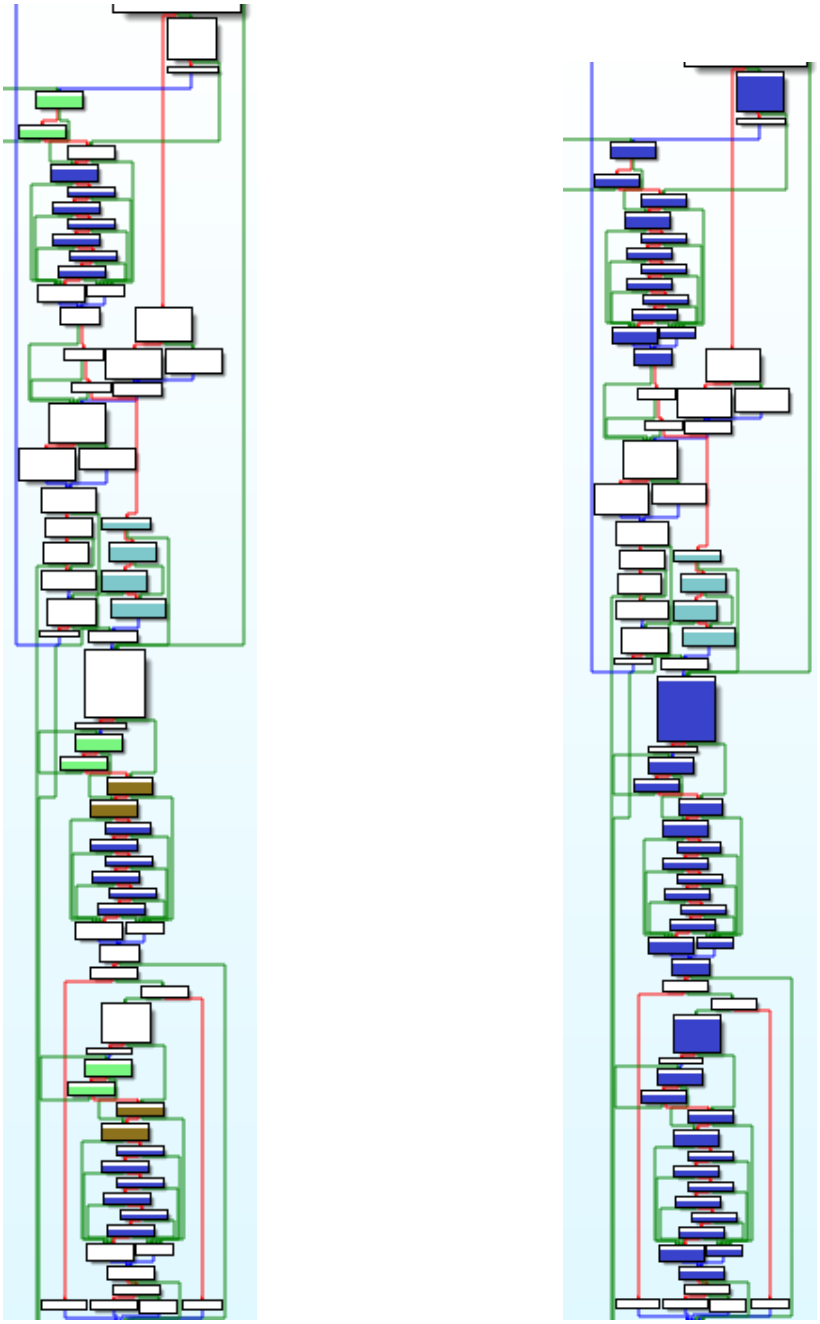
Table 6.2: Average execution time for the analysis process of the program Appendix B.1 compiled with cl, msbuild, and GCC. The analysis time was averaged over 50 runs. The average time and standard deviation are given in the table. The table shows that the time used when the cl compiled version was analyzed was much greater than the rest of the results.

Compiler	**Original GraphSlick**	**Modified GraphSlick**
cl	130 ms ± 1.94 ms	1.79 s ± 68.9 ms
msbuild	19.4 ms ± 624 µs	41.8 ms ± 1.94 ms
GCC	9.06 ms ± 342 µs	28.2 ms ± 1.01 ms

Cl

An unmodified version of GraphSlick was first used to do an initial analysis. The function `test_stringcmp` in Code listing B.4 was analyzed, and the following GraphSlick parameters were used: `minFunctionSizeInBlocks=2` and `hashtype=1`. GraphSlick detected four distinct node groups. Using manual analysis, it was observed that three of them were part of the string-compare function, and one group was unrelated. The modified version of GraphSlick was used to do a subsequent analysis. This analysis resulted in two distinct node groups being identified. Figure 6.2 shows a graphical representation of the results by both the modified and unmodified GraphSlick versions. The full results are in Appendix C.1 in Table C.1.

Figure 6.2 illustrates the analysis of the function with the three instances of the inlined string-compare function. Notably, the analysis by the original GraphSlick resulted in the identification of only the green and blue colored groups in the first instance (Figure 6.2a). However, a brown group was also identified alongside the green and blue groups in the second and third instances. In contrast, the modified GraphSlick identified all three string-compare functions in their entirety in all three instances (Figure 6.2b).



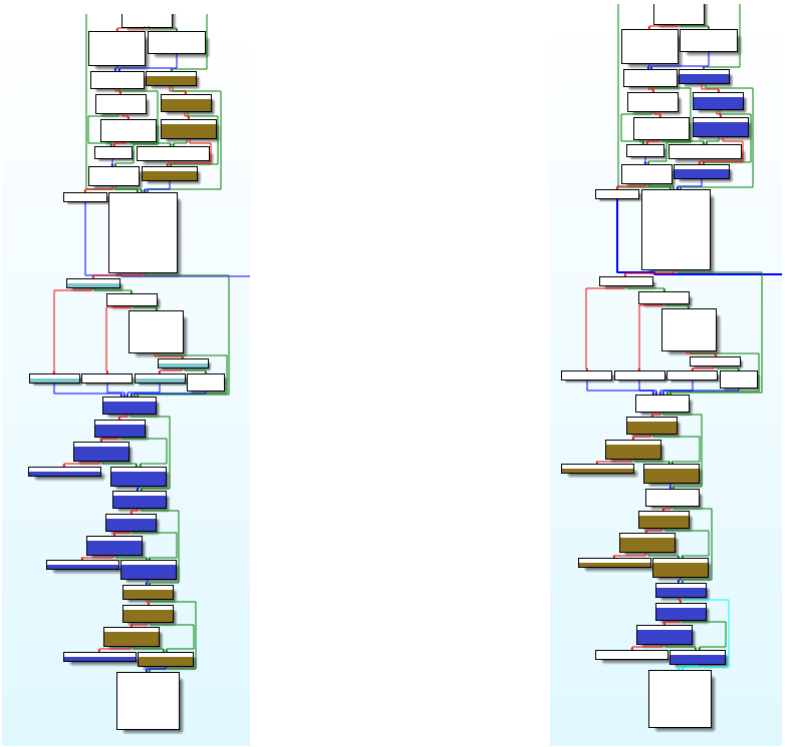
(a) Analysis by original GraphSlick

(b) Analysis by modified GraphSlick

Figure 6.2: Comparison of analysis done on cl compiled program using original and modified GraphSlick. The original GraphSlick was not able to identify the complete inlined function and instead identified three (green, brown, and blue) parts of the larger inlined function, shown in Figure 6.2a. As seen in Figure 6.2b, the modified GrapSlick was able to identify the first block of the inlined function and use this to create a coherent instance.

Msbuild

The same steps and parameters used in analyzing the cl compiled executable were replicated in the analysis of the msbuild executable. Both the original and modified versions of GraphSlick were used to analyze the program, generating the results shown in Table C.2, and visualized in Figure 6.3.



(a) Analysis by original GraphSlick **(b)** Analysis by modified GraphSlick

Figure 6.3: Comparison of analysis done on msbuild compiled program using original and modified GraphSlick. As seen in Figures 6.3a and 6.3b, there was no large difference between the two GraphSlick versions during the analysis. The color difference is only a visual difference.

The analysis done on the msbuild compiled program showed more similarities than the cl compiled one. The original GraphSlick identified three distinct block groups in the executable, and the modified version of GraphSlick found two groups. A manual analysis of the program was also done. This was done by comparing the source code, the compiled program using IDA's disassembler and decompiler, and the GraphSlick analysis.

The manual analysis of the program aimed to validate the results of the analysis done by GraphSlick. It revealed that none of the three groups identified by the first analysis were related to the string compare function. Instead, all the blocks identified were either related to cleanup, exception handling, or just small sections of blocks that coincidentally were similar.

Further manual analysis showed that the string-compare function was inlined differently than in the cl-compiled version. As shown in Code listing 6.1, string compare was inlined, but not in the same way as in the cl version. The code in the listing was taken from the basic block at address *0x140001478*. The basic block at this address contained the inlined string compare function. The string compare function was used, and as shown, there is no reference to a *compare*-function or an unknown function. This indicates that the function was inlined. However, the actual comparison in the string compare function was made using a `memcmp`-function. This function was not inlined, and a call to the `memcmp` may be seen at the fourth last line in Code listing 6.1. No `memcmp` function was found in the cl compiled version. The cl version did not call any functions in the inlined blocks, the blocks detected as duplicates, indicating that the actual comparison was also inlined.

Code listing 6.1: Exerpt from msbuild-compiled string-compare

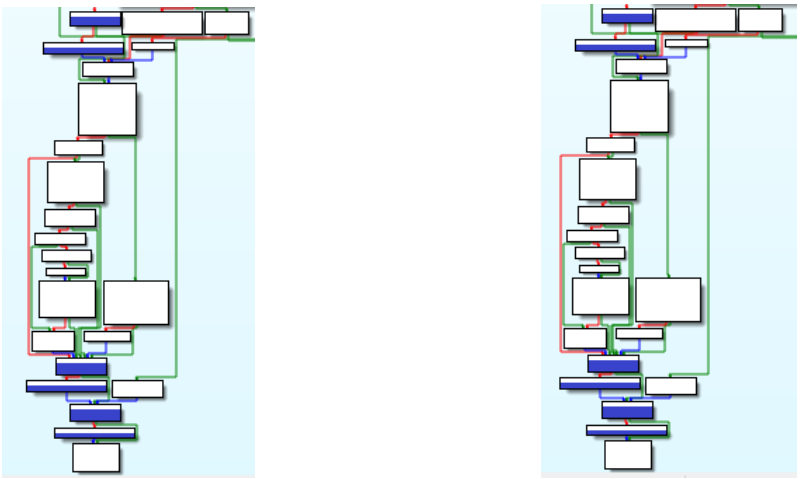
```
                loc_140001478:
4C 8B 75 07     mov     r14, [rbp+57h+Src]
33 C0           xor     eax, eax
48 89 45 E7     mov     [rbp+57h+Buf1], rax
48 89 45 F7     mov     [rbp+57h+var_60], rax
48 C7 45 FF 0F  mov     [rbp+57h+var_58], 0Fh
00 00 00
44 8D 40 0D     lea     r8d, [rax+0Dh]  ; Size
48 8D 15 FF 1E  lea     rdx, aTestString ; "Test & string"
00 00
48 8D 4D E7     lea     rcx, [rbp+57h+Buf1] ; void *
E8 0E 02 00 00  call    sub_1400016B0
48 8D 55 E7     lea     rdx, [rbp+57h+Buf1]
4C 8B 6D E7     mov     r13, [rbp+57h+Buf1]
48 83 7D FF 10  cmp     [rbp+57h+var_58], 10h
49 0F 43 D5     cmovnb  rdx, r13        ; Buf2
48 8D 4D C7     lea     rcx, [rbp+57h+Buf2]
48 8B 5D C7     mov     rbx, [rbp+57h+Buf2]
4C 8B 65 DF     mov     r12, [rbp+57h+var_78]
49 83 FC 10     cmp     r12, 10h
48 0F 43 CB     cmovnb  rcx, rbx        ; Buf1
4C 8B 7D D7     mov     r15, [rbp+57h+var_80]
4D 8B C7        mov     r8, r15
4C 39 7D F7     cmp     [rbp+57h+var_60], r15
4C 0F 42 45 F7  cmovb   r8, [rbp+57h+var_60] ; Size
E8 E7 16 00 00  call    memcmp
8B F8           mov     edi, eax
85 C0           test    eax, eax
75 5C           jnz     short loc_14000153E
```

GCC

The same steps were taken for the GCC compiled program. The full results can be found in the Table C.3, and they are visualized in Figure 6.4. In the analysis of the GCC compiled program, there were no differences in the analysis done by the modified and original versions of GraphSlick. Both versions identified two distinct groups: one group of three instances of two blocks (shown as blue in Figure 6.4)

and a second group of two instances of another two blocks. The second group is not shown in Figure 6.4.

The manual analysis of the program yielded significant findings. Firstly, none of the blocks in the two groups identified were directly related to the string compare functions. Instead, both groups contained smaller code sections associated with cleanup functionality. Secondly, the GCC-compiled program handled string comparison in a manner similar to the msbuild compiled version. No reference to a 'string compare'-function was found in the function. Instead, a call to the function `memcmp` was used, and `memcmp` performed the actual comparison, mirroring the msbuild compiled version.



(a) Analysis by original GraphSlick **(b)** Analysis by modified GraphSlick

Figure 6.4: Comparison of analysis done on GCC compiled program using original and modified GraphSlick. As seen when comparing the colored blocks in Figures 6.4a and 6.4b, there was no difference between the analysis results.

6.2.2 Test 2: Detection of inlined function using compiler keywords

The second test was done using the code and process shown in Appendix B.2. This code uses keywords in the source code to affect which functions the compiler inlines. These keywords are compiler suggestions and are not necessarily enforced all the time, as was described in Sections 2.2.2 and 2.5.2. The keywords accepted by one compiler are not necessarily the same as other compilers. When the code in Code listing B.8 was compiled with cl and msbuild, it was compiled without any modifications to the source code. When GCC was used as the compiler the keywords `__forceinline` and `__declspec(noinline)` in Code listing B.8 were substituted with `__attribute__((always_inline))` and `__attribute__((noinline))`. No other modifications to the code at Code listing B.8 were done regardless of the compiler used.

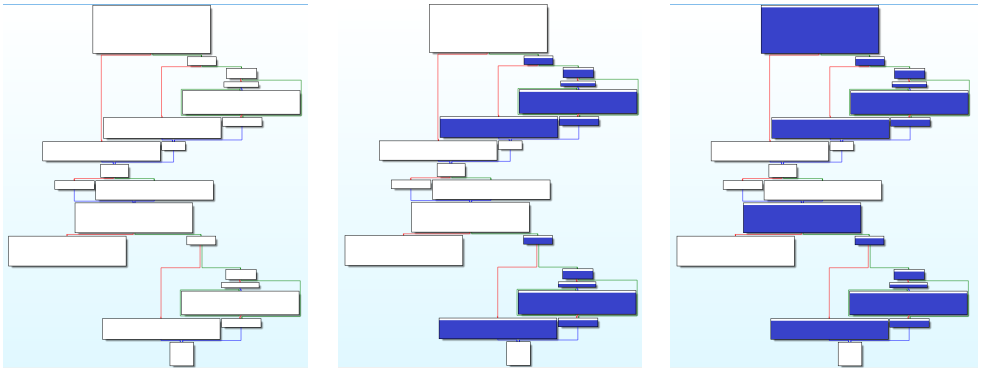
The three instances of the compiled program were all analyzed using both the original and modified versions of GraphSlick. The execution time of the analysis process is shown in Table 6.3.

Table 6.3: Average execution time for the analysis process of the program Appendix B.2 compiled with cl. The analysis time was averaged over 50 loops. The average time and standard deviation are given in the table.

Compiler	**Original GraphSlick**	**Modified GraphSlick**
cl	3.14 ms ± 781 µs	87.1 ms ± 2.06 µs
msbuild	2.15 ms ± 221 µs	39.9 ms ± 342 µs
GCC	1.84 ms ± 184 µs	38.9 ms ± 129 µs

Cl, msbuild & GCC

The code in Code listing B.8 was compiled with cl using the command line in Code listing B.5. The compilation result and analysis results are shown in Figure 6.5. The *main* function in this program had the CFG in Figure 6.5a. Both the original GraphSlick and the modified method identified two instances of one block group when analyzing the program. The modified version identified two extra blocks that were added to their respective block groups, as shown in Table C.4. The original analysis method is visualized in Figure 6.5b and the modified in Figure 6.5c. This was confirmed by manually analyzing the three programs.



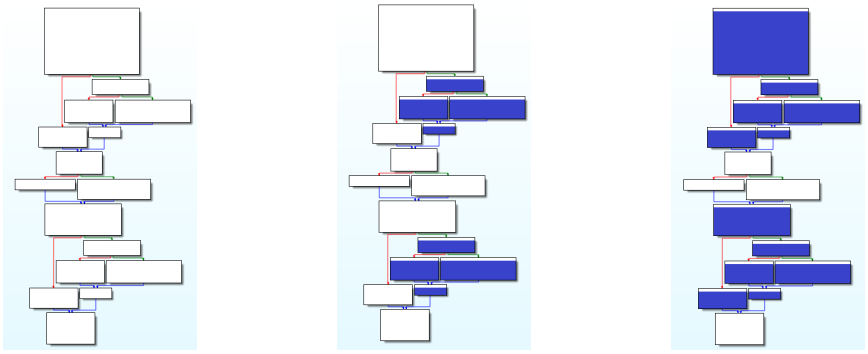
(a) Control flow graph of the program

(b) Analysis done by original GraphSlick

(c) Analysis done by modified GraphSlick

Figure 6.5: Figure shows the whole CFG of the main function of the compiled code from Code listing B.8, and results from the two analysis methods. The code was compiled using cl and analyzed using original and modified versions of GraphSlick.

The same program, code from Code listing B.8, was compiled with msbuild using the compiler options in Code listing B.6. The analysis results are in the appendix in Table C.5. Figure 6.6 shows the CFG of the compiled program and which blocks were identified by the two analysis methods.



(a) Control flow graph of the program

(b) Analysis done by original GraphSlick

(c) Analysis done by modified GraphSlick

Figure 6.6: Figure shows the whole CFG of the main function of the msbuild compiled code from Code listing B.8, and results from the two GraphSlick analysis methods.

The code from Code listing B.8 was finally compiled with GCC using the command in Code listing B.7. The analysis results are in Table C.6, and the whole process is visualized in Figure 6.7.

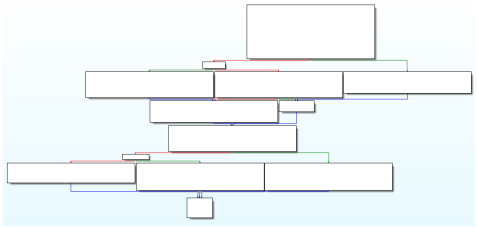
Inlined function using compiler keyword analysis results

The three figures, Figures 6.5 to 6.7, show that the modified version could detect the starting block of the inlined function in all three tests. In the cl compiled version, Figure 6.5, the only difference was the addition of the starting block of the inlined function. The two versions compiled by msbuild and GCC also identified the starting block and additional blocks.

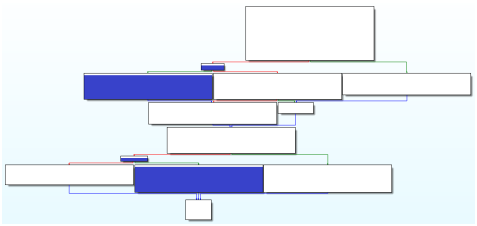
6.2.3 Basic block splitting performance

CFG analysis is a computationally expensive operation. Both the graph creation and the analysis are expensive. When creating the CFG, the hash value for every block must be calculated, and the predecessors and successors of each node must be identified. The analysis of the CFG is also expensive as the program has to go through each block and check if there exists a set of isomorphic graphs based on the starting block. When a basic block is split, the analysis has to be redone.

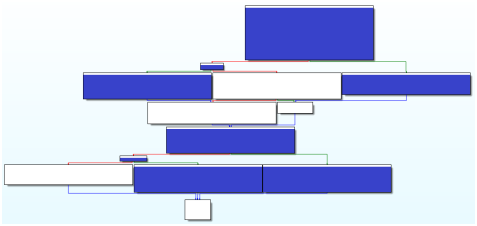
The analysis by the modified GraphSlick version was done on multiple different programs and functions in those programs to explore its performance. Figure 6.8 shows the result of these tests. The programs chosen for the performance



(a) Control flow graph of the program



(b) Analysis done by original GraphSlick



(c) Analysis done by modified GraphSlick

Figure 6.7: Figure shows the whole CFG of the main function of the GCC compiled code from Code listing B.8, and results from the two GraphSlick analysis methods.

test were the ones already analyzed in the previous tests, as well as two common programs. `7zip.dll`[1] and `python311.dll`[2] were used for testing. These two programs were not chosen because of their use of inlined functions. They were instead chosen because they are common programs likely to be representative of programs a reverse engineer could end up analyzing. A reverse engineer does not know if any inlined functions exist in a program when it is initially analyzed, and inlined function detection would be helpful to run regardless of the results.

The tests were timed only by splitting a single block pair. Each data point results from an average of 50 runs of the analysis process. This ensures that the data is more reliable and not as affected by random variables in a single run. As shown in Figure 6.8, the time used on analysis scales with the number of nodes in the graph analyzed. The exact scaling has not been determined. The tests were done up to a CFG with 2061 nodes. Tests with larger graphs were not conducted as the time to complete the analysis was extensive. The whole `7zip.dll` contained 69612 nodes, making it unrealistic to analyze the whole program.

[1] 7zip version 19.0.0, MD5: 72491C7B87A7C2DD350B727444F13BB4
[2] Python version 3.11.4, MD5: 5A5DD7CAD8028097842B0AFEF45BFBCF

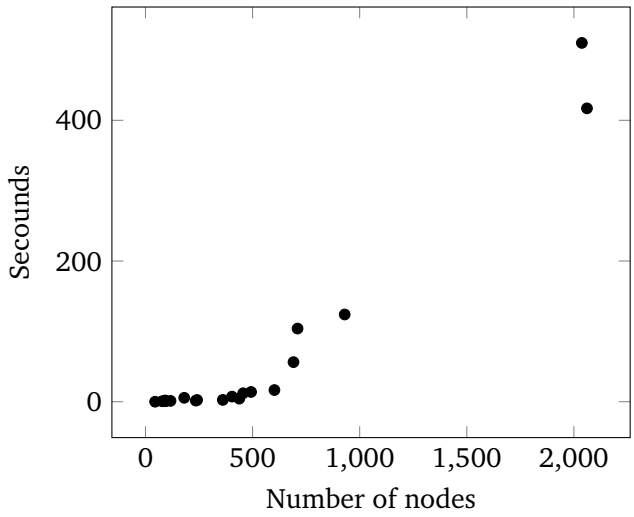


Figure 6.8: The figure shows the time in seconds used when a program was analyzed. The analysis was done using the modified GraphSlick version. The x-axis is the time taken for an analysis to complete. The y-axis is the number of nodes that were analyzed.

6.3 Function identification

To identify a specific inlined function, a signature of a non-inlined version of that function needs to be generated. This signature is a unique identifier for the function and is crucial in the function identification process. It allows a function or parts of a function to be matched with other programs or code parts.

6.3.1 Test 3: Identification of keyword-inlined function

The code from Code listing B.8 was compiled again with keywords changed. The function inlined in the earlier test, described in Section 6.2.2, needed not to be inlined[3]. This made it possible to generate a CFG of that function. To accomplish this, the keyword `__forceinline` was changed into `__declspec(noinline)`. The now not inlined function, the cl compiled version of this function, can be seen in Figure 6.9.

The hashes, GraphSlick type 1 hash, of all basic blocks were generated to compare the not-inlined function with its inlined counterpart. The hashes are listed in Table C.7. This was then compared with the hashes, listed in Table C.8, of the main function in the inlined version of the program. Not all of the nodes in the main function were part of the inlined function. The basic blocks with ID 0 and 2 to 7 were in one group, and blocks with ID 12 and 14 to 19 were in another group. Since these two groups are identical, only the first group, blocks 0 and 2-7,

[3]Not inlined is not the same as outlined. Outlining and inlining are modifications done by the compiler; "not inlining" is just the lack of inlining

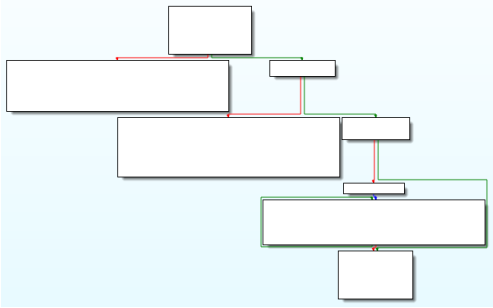


Figure 6.9: Control flow graph of the function that was inlined in the tests done in Section 6.2.2. Using the code from Code listing B.8, without the inline-keyword, compiled by cl using the command shown in Code listing B.5.

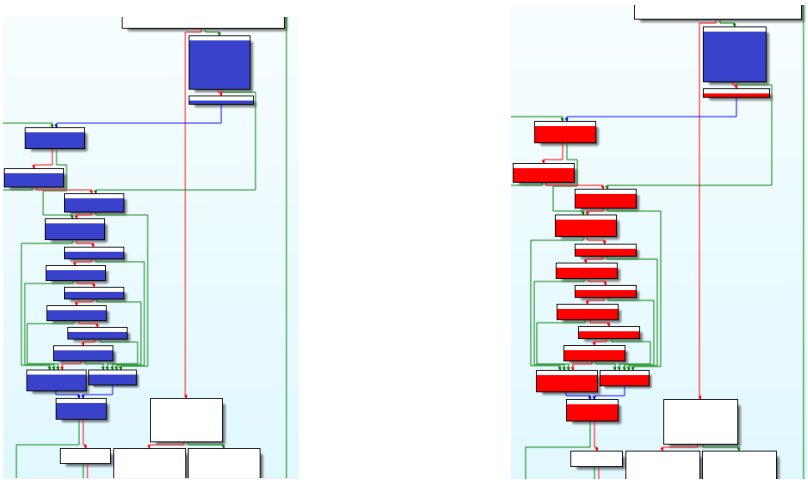
is focused on. Matches were found for nodes with block IDs 2, 5, and 6 in the inlined version of the function, as seen in Tables C.7 and C.8.

Of the seven nodes in the group, only six can realistically find a match since the first block is a "split block" and, therefore, unlikely to have a match in the non-inlined version. The non-inlined version has some stack initialization that the inlined version lacks. Of the blocks that didn't match, two blocks, blocks 1 and 3 in the non-inlined version, included return statements, and the last block, block 4, had one extra instruction, `push rsi`, in the non-inlined version of the function. As `rsi` is a nonvolatile register, this instruction is likely added in the non-inlined version to ensure the register's value is preserved and reclaimed when the function returns. The inlined version does not need to do this, as the concept of volatile and nonvolatile registers does not exist within the context of a single function. The concept of volatile registers is more applicable when there is a caller and callee function.

6.3.2 Test 4: Identification of string comparison function

The function identification test was also done on the program with the inlined string comparison function. This was done on the same source code and compiler settings used in the test described in Section 6.2.1. The test was done to look at the effectiveness of function identification on an inlined function from a standard library as opposed to an inlined function manually inlined from the same source code. This test was only done on the cl-compiled version. The program compiled by other versions created a program where the inlined function comprised a single block, which does not fit a structure suitable for graph analysis. The cl compiled version created a graph with multiple blocks from the inlined function. The string comparison function was compiled as a standalone function to make a reference with which the inlined version could be matched. The version of the string compare function used was `std::string::compare(std::string)`. This function is called the library function. A CFG was created from the library function using hash-type 1.

The program was first analyzed to detect potential inlined functions. Since the program analyzed was the same, and the analysis process is deterministic, this yielded the same result described in Section 6.2.1. The detected inlined functions were then matched with the library function's CFG. The results from this process are shown in Figure 6.10. As shown in Figure 6.10b, all blocks, except the start block, have an equivalent block in the library function. The blocks, block ID 11 to 24, are all hash equal with a block in the library function. All the matches were based on itype 1 hashes generated by GraphSlick. The fact that every block in the inlined function matches the non-inlined version indicates that the entire inlined function was not necessarily detected. Both "starting blocks" and "return blocks" in an inlined function are likely to be affected by compiler optimizations. Since every block, except the first, in the detected inlined function matches the library function, the actual inlined function likely extends to some of the subsequent blocks. This is not a fault with the identification step but relates to the detection step. The only block not found in the library function's CFG is the first block, block 10; this is expected as starting blocks seldom equal their non-inlined counterparts.



(a) Blocks detected as potential inlined code

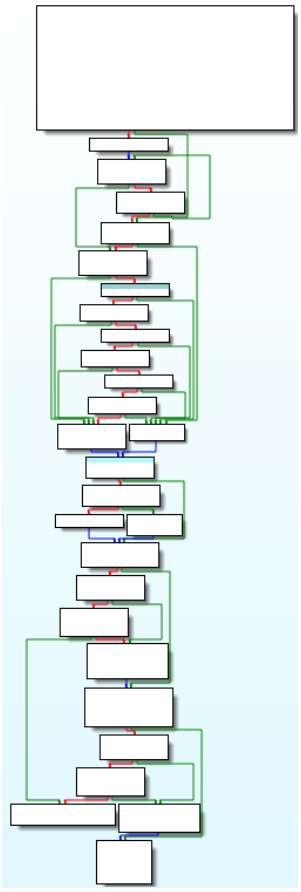
(b) Blocks in the inlined version matching the non-inlined version

Figure 6.10: Figure shows the comparison between the inlined string comparison function and the non-inlined version. The blocks found in the inlined version that are also present in the non-inlined version are colored red in Figure 6.10b.

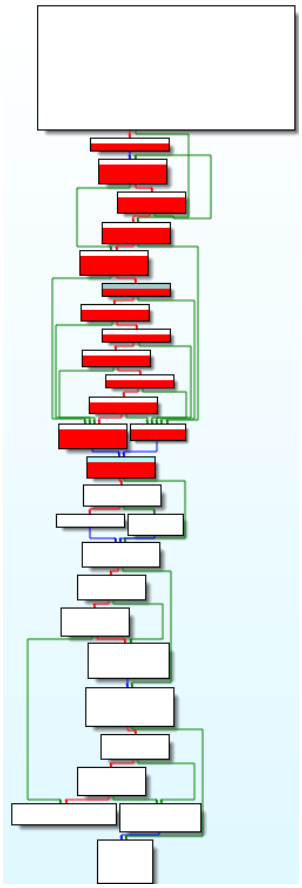
The matching was also done from the opposite perspective: by examining which blocks in the library function have an equal block in the inlined function. This was done using the same method as above, but the functions were switched. The entire library function was used as the "unknown function", and one instance of the detected inlined function was used as the "known function". Since all instances of the inlined function are equal, from a CFG analysis perspective, which instance is used does affect the results.

Figure 6.11 shows the CFG of the compiled function and the blocks that match the inlined version of the function. The start of the function, except the first block, matches the start of the inlined function. This is expected as it should equal the results in Figure 6.10. The end of the function does not match with its inlined counterpart. As described earlier, this indicates that the entire inlined function was not detected, and more blocks exist at the end of the inlined function that are not detected. This is, however, not supported by the results from the detection of the inlined function, shown in Figure 6.1. In this figure, the second and third instances of the inlined function only have two blocks between them, and it would, therefore, not be possible for the inlined function to be as large as its non-inlined counterpart.

A non-inlined version of another string compare function was also tested. This was the same C++ function but with a different argument. The function was `std::string::compare(char*)`. The CFG of the function is shown in Figure 6.12. This function is almost identical to the one in Figure 6.11. However, there are some differences in its beginning. It does also not have as many blocks. This is especially easy to see at the end of the function. As shown in Figure 6.12b, 14 blocks match with its inlined counterpart. This is an equal number as in `std::string::compare(std::string)` version. The equal amount of matching blocks would make it very difficult to decide which of the two library functions the inlined function is identified as. By looking at the number of non-matching blocks, the `std::string::compare(char*)` version has fewer nonmatching blocks, 9 to 11 in the `std::string` version.

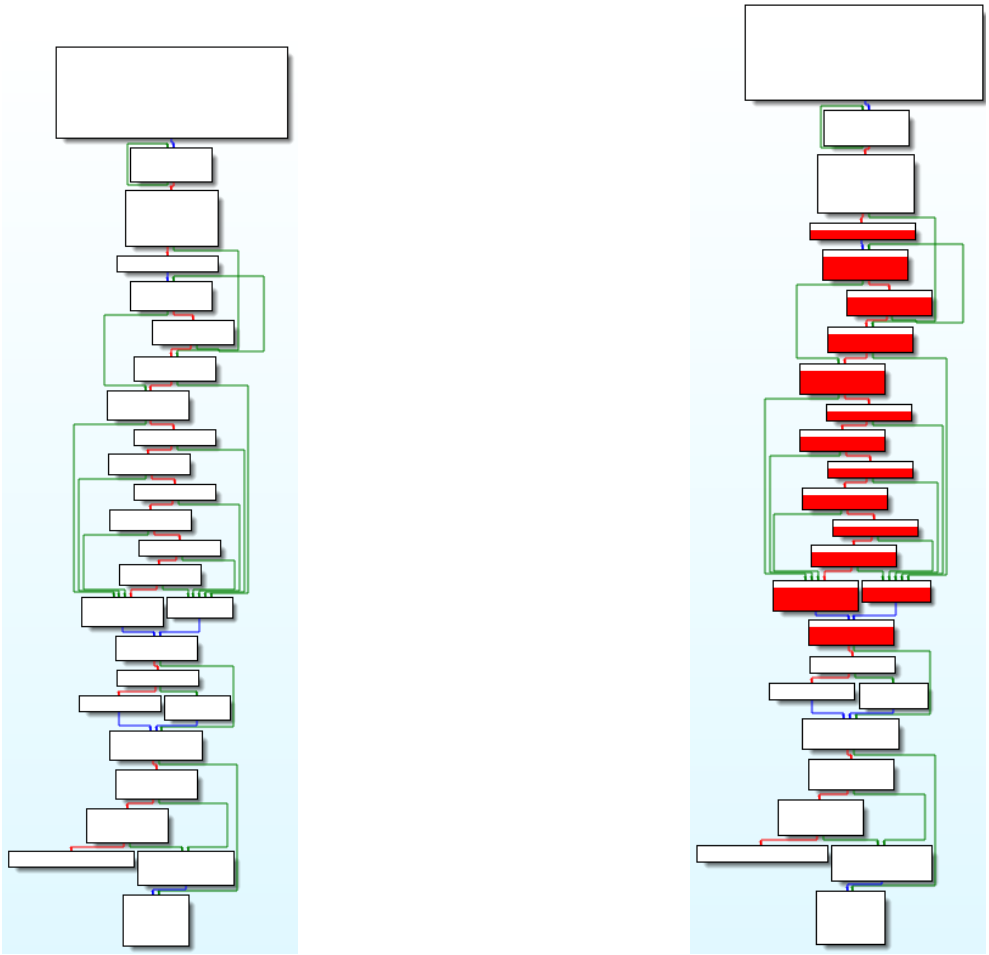


(a) Graph view of the non-inlined function



(b) Blocks from the non-inlined function matching the inlined version

Figure 6.11: Figure shows the graph view of the non-inlined string comparison function with a *std::string* as argument. The blocks found in the inlined version that are also present in the non-inlined version are colored red in Figure 6.11b.



(a) Graph view of the non-inlined function

(b) Blocks from the non-inlined function matching the inlined version

Figure 6.12: Figure shows the graph view of the non-inlined string comparison function with a *char** as argument. The blocks found in the inlined version that are also present in the non-inlined version are colored red in Figure 6.12b.

Chapter 7

Discussion

This chapter discusses the results from the previous chapter and their implications. It discusses inlined function detection and identification separately. It also covers previously unforeseen problems discovered during testing, like single-block inlining.

7.1 Validating the method

Validating the effectiveness of the tests in function inlining presents several challenges, primarily revolving around creating a comprehensive dataset that accurately represents the behavior and structure of functions when inlined. In the test shown in Section 6.3.2 from the perspective of the inlined function, it matches quite well with the non-inlined version. The non-inlined version does, however, have a lot of extra blocks that do not appear in the inlined version of the function. These blocks relate to stack cleanup, register preservation, exception handling, C++ object deallocation, and returning the correct value. This is likely optimized away in the inlined version; exception handling and memory cleanup are handled as a whole for the entire function and not separately by each instance of the inlined function.

Validating if a function match is identified is difficult since the results differ based on the perspective. An inlined version of a function is optimized compared to its non-inlined counterpart. Only half the blocks in the non-inlined functions were a match. Another problem is in cases where there are different versions of the same function as described in Section 4.3.3. In those cases, it would likely not be possible to differentiate the functions using the CFG analysis method tested in this thesis. If differentiating between them is important depend on the situation, and there are cases where there would be little to be gained in differentiation between them. Regardless of the potential lack of gain, this is a weakness of the method and should, therefore, be highlighted or further explored to improve it.

Another problem with verification is how compiler optimization, dead code removal, runtime setup, and library code lead to the detection of false negatives. Identifying the number of false negatives poses a challenge. The use of object files

to identify if a function has been removed by optimizations could assist in this. However, it has not been extensively tested in an automated setting.

7.2 Known functions signature database

When compiling source code into a program, the structure of the resulting program is highly dependent on the compiler and compiler options used. The CFGs of the programs shown in Figure 6.1 are compiled from the same source code. Some overall structures are similar in the CFGs, but several differences make it challenging to identify that the graphs are from the same source code. Some differences could be attributed to different compilers using different implementations of the standard library to compile a program. This is not the only difference since significant differences exist between the two programs compiled with cl and msbuild, both of which use the Microsoft Visual Studio toolchain. Different compilation toolchain uses different implementations for c standard libraries; gcc uses *glibc*. In contrast, the Microsoft compilers use the *C runtime*, also called *CRT*, and clang uses *libc*. Different standard libraries differ in their implementation of low-level functionality, like how they handle C++ objects. This could partially explain the significant difference in CFGs by different compilers. Differences in the programs compiled in Figures 6.5 to 6.7 also show that the differences are not only connected to the implementation standard functions. In *Test 2* described in Section 6.2.2, the inlined function did not come from a standard library. Since the inlined function was from the compiled program source code, the implementation of the inlined function was not different across compilers. This indicates that a program's CFG depends highly on the compiler and compiler flags used.

The method proposed in this thesis for function identification depends on having a dataset of the known CFGs of known non-inlined versions of functions. To match a potentially inlined function with a known function in the dataset, the dataset must encompass various versions of the known function compiled with different compilers and compiler flags. This diversity in the dataset is crucial for the method's effectiveness in real-world scenarios. A dataset containing diverse compiler keyword and flag combinations enhances the robustness and universal capabilities of the function identification method. Having multiple versions of known functions compiled with different compilers, compiler keywords, and flags allows for a broader coverage of possible code transformations and optimizations. It ensures that the method is not overspecialized to a particular compiler or optimization setting, making it more applicable to real-world software systems developed where various tools and environments are used. However, achieving this diversity in the dataset could be challenging and require significant resources and time.

The reliance on compiling a known function with different compilers and the lack of robustness across compilers can be seen as a weakness of the method. This problem could create situations where a match is not found because the dataset isn't extensive enough. By only using mnemonics when matching and detect-

ing functions, there is some tolerance to compiler differences, and the method is somewhat resistant to situations where compiler optimizations make distinct instances of a function differ across a program. The tolerance and resistance are, however, not great, and the method would still need many different signatures of the same function to be reliably able to identify it. This is not a problem and weakness unique to this method. Other industry-standard methods and tools for detecting and identifying something, like IDA Pro's FLIRT signatures and YARA rules, are equally affected by compiler differences. Both FLIRT and YARA use binary matching to identify something; because of this, they are affected by the differences between compilers. The binary matching also makes them somewhat less robust compared to the method presented in this thesis. By using mnemonics instead of opcodes, an operation like `mov rax, [rsp]` is not differentiated from `mov rax, rbx`. Not differentiating between these operations is not necessarily a good or bad thing. In some cases, they might be equivalent, and the difference is created by the compiler, whilst in other cases, not differentiating between them is an error. FLIRT and YARA work on a binary level and achieve some robustness by allowing some series of bytes to be anything. This is achieved by using wildcards. Support for wildcards allows for some flexibility in the signatures and rules, but they are generally still more strict than using mnemonics.

Creating an extensive dataset containing the necessary number of known functions would be difficult in the short term. Initially, a lot of data would be needed to get started. To make the method viable as quickly as possible, it would be sensible to do research into the most common compilers and compiler flags used. Finding the most common libraries used in projects where the functions will likely be inlined and optimized will also be sensible.

In the long term, the large amount of data needed might pose a problem. A big dataset would significantly increase the time needed for function identification. A larger dataset would include more functions that are potential matches and several versions of the same function, all of which would have to be tested. This could lead to scalability issues, as the time and resources required for function identification would increase with the size of the dataset. To manage these issues, it would be sensible to limit the number of known functions needed to be tested against. One way to do this would be to use compiler identification. Labeling the known functions in the dataset with the compiler and compiler options used, when creating that known function, would make it possible to only match with functions from a specific compiler. If compiler identification were done on the program being analyzed, this would limit the need to match against known functions from other compilers. Depending on the precision of the compiler identification, it might be possible to limit the matching process to only lookup known functions with specific compiler flags. The accuracy and robustness of compiler identification tools must be validated before this can be implemented.

Identifying the correct match with certainty is difficult, as discussed in Section 7.1. As seen in the results from Figure 6.9, the same function compiled on the exact same system differs between an inlined version and a non-inlined ver-

sion. This makes it challenging to identify whether a specific known function best matches an unknown inlined function. Without the ability to be sure if the correct match is found, it will be unfeasible to optimize the process by stopping the identification process halfway through the dataset. The identification process must go through the entire dataset to find the best-matching known function.

7.3 Detecting functions

The current tests were done by detecting duplicate code in a single function. The program analyzed included multiple instances of a function, function B, being inlined in the same function, function A. The analysis could be done across various functions. Nothing in the analysis method depends on the CFG being from a single function or that the CFG needs to be comprised of a single connected graph. The original version of GraphSlick could not support this because it used block IDs as a key when referencing blocks in the CFG. Block IDs overlap across different functions. If GraphSlick used a block's starting address or the address of the start of the function added to the block IDs, it would be able to analyze multiple functions and detect duplicate code across them. Using a substitute for a key in the CFG, instead of block IDs, that is unique across the entire program would not affect IDA. Since GraphSlick uses its own CFG, the rest of IDA is unaffected by changes in the structure of the CFG.

Analyzing the entire program would be helpful and necessary in real-world use cases. The code detection depends on the assumption that more than one instance of the function being inlined exists. If this assumption is correct has yet to be evaluated, and this thesis has not done tests to validate it. The original GraphSlick version relied on the duplicate code being in a single function to detect it. In reality, functions can be reused across different parts of the program, leading to instances of an inlined function scattered throughout various functions, modules, or multiple files. Therefore, a more comprehensive approach that considers the entire program rather than individual functions in isolation is likely to yield more accurate results.

7.3.1 Non-duplicated inlined code

A weakness of the method used in this thesis for inlined function detection is that it doesn't actually detect inlined functions. Instead, it detects duplicate code with a structure that is potentially from an inlined function. This relies on the assumption that the code is inlined in multiple locations. Like all assumptions, it might not be correct. As previously described, this thesis has yet to find a method to differentiate between machine code from an inlined function and other sources. Detecting a single instance of an inlined function is impossible with the technique described in this thesis.

7.3.2 Detecting single block inlined functions

Test 1, as described in Section 6.2.1, showed that the method is ineffective at identifying some inlined functions. The analysis could not detect the inlined function when the program was compiled using msbuild or gcc. As described earlier, this can be attributed to the analysis process not detecting inlined functions but duplicate series of basic blocks. When an inlined function was inlined in a single basic block, the CFG analysis did not work since there wasn't a graph with multiple nodes to analyze. The work done in this thesis has not identified a solution to match single-block inlined functions. Possible solutions to this problem have been considered, but they have been found ineffective. One solution that has been considered is to set `minFunctionSizeInBlocks=1` the analysis. This would enable the analysis to match singular equal blocks. The problem with this is that it would create a lot of false positives, matching blocks that are not related to an inlined function. It would also not necessarily be able to match single-block inlined functions since the single block would be the first block of that inlined function. As previously described, when inlining a function, the first block of the inlined function contains code from both the function being inlined into and the inlined function. Block splitting would, therefore, also be necessary to match single-block inlined functions. This makes detecting single-block inlined functions unfeasible. Without any initial analysis indicating the potential block-splitting candidates, the analysis process would have to try to split and match any two blocks. This would generate more false positives and be prohibitively computationally expensive. Other methods are better suited to detect code similarities between single blocks. CFG analysis is more suitable when multiple nodes are compared.

7.3.3 Performance

The performance of the analysis could be a limiting factor. The time taken by the analysis process depends on the number of blocks contained in the CFG analyzed. The tests show that the analysis time does not scale linearly with the number of blocks. The exact performance parameters of the analysis process are not fully explored, but based on the test results, it is probably somewhat exponential.

A relatively simple function, like the function analyzed in Section 6.2.1 compiled by cl, contained 98 blocks. The average analysis time for this function was 1.79 seconds using the modified GraphSlick. Analyzing a CFG with 2037 nodes took 8 minutes and 30 seconds. An analysis across all functions in a more extensive program would be unfeasible in a reasonable time as programs could have tens of thousands of blocks.

An analysis of a CFG of an entire program would seldom be needed. Other faster methods for function identification could be used in conjunction with the CFG-based method. If one of the quicker methods, like IDA's FLIRT-signatures, identifies a function, adding the CFG of this function to the CFG being analyzed would not be needed. This assumes that other analysis methods correctly identify functions. Given this assumption, it implies that this function identified by another

technique is a known function. Therefore, further code analysis with the purpose of function identification is redundant. Functions relating to program start are also candidates to be excluded from a CFG analysis of the programs. Functions related to the initialization process, such as the initialization of the c-runtime library, create a lot of blocks that relate to standard libraries. If blocks from these functions are excluded from the CFG, a higher proportion of the CFG would contain blocks relevant to the analysis of the program's functionality.

7.4 Test environment

The tests were only conducted on a limited number of test environments. Only Windows was used as an OS, and the number of compilers was limited. As seen in the function identification tests, quite small differences in functions, like the arguments used to call it, could create large differences in the CFG of the output function. The same was seen when the same program was compiled with differing compilers. This indicates that further testing with another OS, different CPU architecture, and other compilers would be beneficial.

The tests did, on the other hand, reveal that the method did not function differently when tested with a differing set of compilers. In all cases where the method was expected to perform it did so to a certain degree. In the situations where it didn't perform, like with the single-block inlined functions, this was expected as CFG analysis requires multiple blocks in a CFG. The reason for its failure was, therefore, not related to any compiler or environment setting. Instead, it was related to a specific implementation of a function and how it was inlined in the standard library used by the compiler.

Chapter 8

Conclusion

This project has demonstrated that Control Flow Graph (CFG) analysis is a viable technique for detecting and identifying inlined functions. CFG analysis has proven effective in discerning the structures and characteristics of some types of inlined functions. The results suggest that CFG analysis can be integrated into existing reverse engineering frameworks to enhance their capabilities in handling inlined functions. The project has built and expanded upon the work done in GraphSlick.

GraphSlick previously showed that CFG analysis can be used to detect duplicate code. This thesis's advancement to the GraphSlick project showed that duplicate code detection can also be used to detect inlined functions. This was done by splitting basic blocks. A single basic block can include code from different parts of a function. By dividing the top from the bottom of a basic block, the bottom creates a possible starting point from which a function-like graph can be derived. The method proposed in this thesis can only detect inlined functions that contain more than one basic block. In general, CFG analysis depends on a graph containing multiple blocks that can be analyzed to be effective. This limitation is not only applicable to the implementation chosen by this thesis.

Inlined function identification was also explored. Analysis showed that an exact match between a known function and an inlined instance of that function is not feasible. Even without an exact match, it was possible to identify an inlined function with its non-inlined counterpart. This was done by creating a CFG of the inlined function and comparing it with the CFG of known functions that potentially match. Problems relating to the "entry point" and return blocks of a function when inlined were discussed, and possible solutions were explored.

A framework for testing and validating function detection and identification techniques was visualized in Figure 5.1. It described and explored the necessary components in a testing system designed to verify the analysis of inlined functions. The main components of a testing framework are the test data generation and the verification. Generating good test data and verifying results is challenging because of the complexities in the compiler and the linker. These challenges can be overcome by using intermediary steps in the compilation process to generate verification data.

To summarize, this thesis expanded on the current reverse engineering knowledge related to function identification. It focused on detecting and identifying inlined functions and showed that CFG analysis is a viable technique for this task, which should be explored further.

8.1 Future Work

While showcasing the versatility of CFG analysis in tackling complex problems in reverse engineering, the thesis also underscores the need for further research. It has laid the groundwork and shown promising results at an initial stage, but more work is still needed before it can be used and deployed as more than a proof of concept. This presents an opportunity for other researchers to contribute to the advancement of techniques related to inlined code detection and identification.

Any method of function identification depends on a dataset of known functions to which unknown functions can be matched. The dataset needs to be extensive to identify unknown functions, as the origin of the unknown function is not necessarily known. As the thesis has focused on creating a proof of concept, it has yet to examine how a dataset of library functions could be made at scale. The dataset's storage, distribution, and updating must also be explored. Similar datasets, like IDA's FLIRT signatures, could be used as an inspiration in this process.

The technique was only tested on a single operating system, Windows, with a limited number of compilers and programs. More extensive tests would need to be done to establish greater faith in the technique. Testing with a larger sample size of compilers, compiler flags, and programs would also help better understand its strengths and limitations. The thesis has described the design and implementation of a testing framework for more extensive tests. Tests comparing different techniques, like the one using execution flow graphs proposed by Qiu *et al.* [83], should also be explored.

Further research is needed to investigate the performance of CFG analysis, particularly in the context of large graphs. While the tests conducted in this project validate the effectiveness of CFG analysis for inlined function detection, they also reveal significant scalability issues. The current method's performance degrades as graph size increases, leading to longer processing times and higher computational resource consumption. Future work should focus on optimizing the CFG analysis algorithm to handle larger graphs more efficiently. Potential areas for improvement include algorithmic enhancements, parallel processing techniques, and graph reduction techniques. Addressing these performance bottlenecks is crucial for making CFG analysis a practical tool for real-world applications involving extensive codebases.

Bibliography

[1] S. Mansfield-Devine, 'The ashley madison affair,' *Network Security*, vol. 2015, no. 9, pp. 8–16, 2015.

[2] A. Greenberg, 'The untold story of notpetya, the most devastating cyberattack in history,' *Wired, August*, vol. 22, 2018.

[3] O. E. Dictionary, *reverse-engineer, v* (Oxford English Dictionary). Oxford University Press, Jul. 2023.

[4] D. A. Solomon, M. E. Russinovich and A. Ionescu, *Windows internals*. Microsoft Press, 2009.

[5] A. Alfred V, L. Monica S, S. Ravi, U. Jeffrey D *et al.*, *Compilers-principles, techniques, and tools*. pearson Education, 2007.

[6] The FreeBSD Project, *Clang tools documentation*, `https://man.freebsd.org/cgi/man.cgi?query=clang-cpp&sektion=1&manpath=FreeBSD+9.0-RELEASE`, [Accessed 10-05-2024].

[7] The Clang Team, *Clang - the clang c, c++, and objective-c compiler*, `https://clang.llvm.org/docs/CommandGuide/clang.html`, [Accessed 26-03-2024].

[8] The Clang Team, *Llvm language reference manual*, `https://llvm.org/docs/LangRef.html`, [Accessed 26-03-2024].

[9] R. Awati, *Assembler*, `https://www.techtarget.com/searchdatacenter/definition/assembler`, [Accessed 03-05-2024].

[10] Microsoft, */ltcg (link-time code generation)*, `https://learn.microsoft.com/en-us/cpp/build/reference/ltcg-link-time-code-generation`, Accessed: 2023-10-02, 22nd Sep. 2022.

[11] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang and M. Debbabi, 'Bincomp: A stratified approach to compiler provenance attribution,' *Digital Investigation*, vol. 14, S146–S155, 2015.

[12] Z. Tian, Y. Huang, B. Xie, Y. Chen, L. Chen and D. Wu, 'Fine-grained compiler identification with sequence-oriented neural modeling,' *IEEE Access*, vol. 9, pp. 49 160–49 175, 2021. DOI: `10.1109/ACCESS.2021.3069227`.

[13] M. Poslušný and P. Kálnai, *Vb2019 paper: Rich headers: Leveraging this mysterious artifact of the pe format*, https://www.virusbulletin.com/virusbulletin/2020/01/vb2019-paper-rich-headers-leveraging-mysterious-artifact-pe-format/, [Accessed 28-04-2024].

[14] Microsoft, *Pe format*, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format, [Accessed 28-04-2024].

[15] Bytepointer, *The undocumented microsoft "rich" header*, https://bytepointer.com/articles/the_microsoft_rich_header.htm, [Accessed 28-04-2024].

[16] A. D. Forfot, 'Exploring the pe header and the rich header for effective malware classification and triage,' M.S. thesis, NTNU, 2021.

[17] M. E. Russinovich, D. A. Solomon and A. Ionescu, *Windows internals, part 2*. Pearson Education, 2012.

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

[19] The 86open Project, *Unix-on-intel players agree on a common binary (it's the linux elf format)*, https://web.archive.org/web/20070227214032/http://www.telly.org/86open/, [Accessed 26-04-2024].

[20] Statcounter, *Desktop operating system market share worldwide*, https://gs.statcounter.com/os-market-share/desktop/worldwide, [Accessed 03-05-2024].

[21] A. Sherif, *Global market share held by operating systems for desktop pcs, from january 2013 to february 2024*, https://www.statista.com/statistics/218089/global-market-share-of-windows-7/, [Accessed 03-05-2024].

[22] Fortune Business Insights, *Server operating system market volume, share & industry analysis, by operating system (windows, linux, unix, and others)*, https://www.fortunebusinessinsights.com/server-operating-system-market-106601, [Accessed 03-05-2024].

[23] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012, ISBN: 978-1593272906.

[24] K. Gupta and T. Sharma, 'Changing trends in computer architecture: A comprehensive analysis of arm and x86 processors,' *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 7, 2021.

[25] A. Fog, 'The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers,' *Copenhagen University College of Engineering*, vol. 3, May 2023.

[26] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, 'Spectre attacks: Exploiting speculative execution,' in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, 'Meltdown: Reading kernel memory from user space,' in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[28] Intel, 'Intel® 64 and IA-32 Architectures Software Developer's Manual,' *Volume 1: Basic Architecture*, vol. 1, 2023.

[29] J. F.
bibinitperiod f. Dean Elsner, *80386 dependent features - at&t syntax versus intel syntax*, `https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_16.html`, [Accessed 02-11-2023], 1994.

[30] vivek, *At&t assembly syntax*, `https://csiflabs.cs.ucdavis.edu/~ssdavis/50/att-syntax.htm`, [Accessed 02-11-2023], 2003.

[31] Intel, *Eflags cross-reference and condition codes*, `https://www.cs.utexas.edu/~byoung/cs429/condition-codes.pdf`, [Accessed 28-04-2024].

[32] T. Shanley, *Protected mode software architecture*. Taylor & Francis, 1996.

[33] F. Cloutier, *Jcc — jump if condition is met*, `https://www.felixcloutier.com/x86/jcc`, [Accessed 28-04-2024].

[34] Intel, 'Intel® 64 and IA-32 Architectures Software Developer's Manual,' *APPENDIX B EFLAGS CONDITION CODES*, vol. 2, 2023.

[35] A. Fog *et al.*, 'Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus,' *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011, Updated version at: `https://www.agner.org/optimize/instruction_tables.pdf`, Accessed 2023-10-02.

[36] Intel, 'Intel Analysis of Speculative Execution Side Channels,' 2018.

[37] Stanford CS107 joint effort, *Guide to x86-64*, `https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_x86-64.html`, [Accessed 03-05-2024].

[38] *Inline functions (c++)*, `https://learn.microsoft.com/en-us/cpp/cpp/inline-functions-cpp`, Accessed: 2023-10-02, 25th Aug. 2022.

[39] Free Software Foundation, Inc, *3.11 options that control optimization*, `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, [Accessed 27-05-2024].

[40] Microsoft, */ob (inline function expansion)*, `https://learn.microsoft.com/en-us/cpp/build/reference/ob-inline-function-expansion?view=msvc-170`, [Accessed 08-05-2024].

[41] *Llvm weekly - #186, jul 24th 2017*, `https://llvmweekly.org/issue/186`, Accessed: 2023-10-02, 24th Jul. 2017.

[42] M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, 'Comprehensive shellcode detection using runtime heuristics,' in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 287–296.

[43] A. Obregon, *An introduction to java bytecode*, https://medium.com/\spacefactor\@m{}AlexanderObregon/an-introduction-to-java-bytecode-885677548674, [Accessed 27-03-2024].

[44] VMProtect Software, *Vmprotect overview*, https://vmpsoft.com/vmprotect/overview, [Accessed 27-03-2024].

[45] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach* (ISSN). Elsevier Science, 2011, ISBN: 9780123838735.

[46] Microsoft, *Exitprocess function (processthreadsapi.h)*, https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess, [Accessed 26-03-2024].

[47] OSIRIS Lab, *Return oriented programming*, https://ctf101.org/binary-exploitation/return-oriented-programming/, [Accessed 26-03-2024].

[48] R. Meier, *Reverse engineering introduction to the world of disassembling and decompiling*, https://www.scip.ch/en/?labs.20211202, [Accessed 04-12-2023].

[49] HexRays, *Introduction to decompilation vs. disassembly*, https://hex-rays.com/decompiler/decompilation_vs_disassembly/, [Accessed 01-12-2023].

[50] F. E. Allen, 'Control flow analysis,' *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[51] D. Bruschi, L. Martignoni and M. Monga, 'Detecting self-mutating malware using control-flow graph matching,' in *Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3*, Springer, 2006, pp. 129–143.

[52] G. Bonfante, M. Kaczmarek and J.-Y. Marion, 'Control flow graphs as malware signatures,' in *International workshop on the Theory of Computer Viruses*, 2007.

[53] M. S. Håland, 'Multinomial malware classification using control flow graphs,' M.S. thesis, NTNU, 2019.

[54] M. O. Østbye, 'Multinomial malware classification based on call graphs,' M.S. thesis, NTNU, 2017.

[55] S. Näher, 'Leda, a platform for combinatorial and geometric computing,' in *Handbook of Data Structures and Applications*, Chapman and Hall/CRC, 2018, pp. 653–666.

[56] European Molecular Biology Laboratory, *Graph theory: Graph types and edge properties*, https://www.ebi.ac.uk/training/online/courses/network-analysis-of-protein-interaction-data-an-introduction/introduction-to-graph-theory/graph-theory-graph-types-and-edge-properties/, [Accessed 27-03-2024].

[57] X. Hu, T.-c. Chiueh and K. G. Shin, 'Large-scale malware indexing using function-call graphs,' in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 611–620, ISBN: 9781605588940. DOI: 10.1145/1653662.1653736. [Online]. Available: https://doi.org/10.1145/1653662.1653736.

[58] B. G. Ryder, 'Constructing the call graph of a program,' *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[59] C. Cifuentes and M. Van Emmerik, 'Recovery of jump table case statements from binary code,' *Science of Computer Programming*, vol. 40, no. 2-3, pp. 171–188, 2001.

[60] D. Grove, G. DeFouw, J. Dean and C. Chambers, 'Call graph construction in object-oriented languages,' in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1997, pp. 108–124.

[61] L. Zobernig, S. D. Galbraith and G. Russello, 'When are opaque predicates useful?' In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, IEEE, 2019, pp. 168–175.

[62] Intel, 'Intel® 64 and IA-32 Architectures Software Developer's Manual,' *Volume 2A: Instruction Set Reference, M-U*, vol. 2, 2023.

[63] S. Romano, C. Vendome, G. Scanniello and D. Poshyvanyk, 'A multi-study investigation into dead code,' *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 71–99, 2018.

[64] R. Naug and D. kavita, 'A study of types of dead codes and their solutions,' *International Journal of Advance Research in Computer Science and Management*, Apr. 2021.

[65] J. Koret, *A new control flow graph based heuristic for diaphora*, https://github.com/joxeankoret/diaphora, Accessed: 2024-05-27.

[66] HexRays, *Hex Rays - State-of-the-art binary code analysis solutions*, https://hex-rays.com/ida-pro/, [Accessed 31-10-2023].

[67] C. Karamitas and A. Kehagias, 'Efficient features for function matching between binary executables,' in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2018, pp. 335–345.

[68] *A new control flow graph based heuristic for diaphora*, https://joxeankoret.com/blog/2018/11/04/new-cfg-based-heuristic-diaphora/, Accessed: 2023-04-16.

[69] *Bindiff manual*, https://www.zynamics.com/bindiff/manual/, Accessed: 2023-04-16.

[70] M. Arutunian, H. Hovhannisyan, V. Vardanyan, S. Sargsyan, S. Kurmangaleev and H. Aslanyan, 'A method to evaluate binary code comparison tools,' in *2021 Ivannikov Memorial Workshop (IVMEM)*, IEEE, 2021, pp. 3–5.

[71] H. P. Luhn, 'A new method of recording and searching information,' *American Documentation*, vol. 4, no. 1, pp. 14–16, 1953. DOI: https://doi.org/10.1002/asi.5090040104. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/asi.5090040104. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.5090040104.

[72] M. T. McClellan and J. Minker, *The art of computer programming, vol. 3: Sorting and searching*, 1974.

[73] L. Chi and X. Zhu, 'Hashing techniques: A survey and taxonomy,' *ACM Comput. Surv.*, vol. 50, no. 1, Apr. 2017, ISSN: 0360-0300. DOI: 10.1145/3047307. [Online]. Available: https://doi.org/10.1145/3047307.

[74] Daisie Team, *Comprehensive guide to non-cryptographic hash functions*, https://blog.daisie.com/comprehensive-guide-to-non-cryptographic-hash-functions/, [Accessed 26-04-2024].

[75] D. E. Eastlake and P. Jones, *US Secure Hash Algorithm 1 (SHA1)*, RFC 3174, Sep. 2001. DOI: 10.17487/RFC3174. [Online]. Available: https://www.rfc-editor.org/info/rfc3174.

[76] US National Institute of Standards and Technology, *Nist transitioning away from sha-1 for all applications*, https://csrc.nist.gov/news/2022/nist-transitioning-away-from-sha-1-for-all-apps, [Accessed 27-03-2024].

[77] E. Barker and A. Roginsky, *Transitioning the use of cryptographic algorithms and key lengths*. Mar. 2019. DOI: 10.6028/nist.sp.800-131ar2. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-131Ar2.

[78] *F.l.i.r.t.* https://hex-rays.com/products/ida/tech/flirt/, Accessed: 2023-04-15.

[79] *Ida f.l.i.r.t. technology: In-depth*, https://hex-rays.com/products/ida/tech/flirt/in_depth/, Accessed: 2023-04-15.

[80] J. McMaster, *Issues with flirt aware malware*, 2011.

[81] K. Griffin, S. Schneider, X. Hu and T.-c. Chiueh, 'Automatic generation of string signatures for malware detection,' in *Recent Advances in Intrusion Detection: 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings 12*, Springer, 2009, pp. 101–120.

[82] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi and A. Hanna, 'Binsign: Fingerprinting binary functions to support automated analysis of code executables,' in *ICT Systems Security and Privacy Protection*, S. De Capitani di Vimercati and F. Martinelli, Eds., Cham: Springer International Publishing, 2017, pp. 341–355, ISBN: 978-3-319-58469-0.

[83] J. Qiu, X. Su and P. Ma, 'Using reduced execution flow graph to identify library functions in binary code,' *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 187–202, 2016. DOI: 10.1109/TSE.2015.2470241.

[84] J. Qiu, X. Su and P. Ma, 'Library functions identification in binary code by using graph isomorphism testings,' in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 261–270. DOI: 10.1109/SANER.2015.7081836.

[85] A. Rahbar, A. Pezeshk and E. Bachaalany, *Graphslick*, https://github.com/lallousx86/GraphSlick, Commit f7fac52852702a3c7300ba2474c9ab0eda59359b, 2014.

[86] HexRays, *2014 plug-in contest*, https://hex-rays.com/contests_details/contest2014/#graphslick, [Accessed 26-03-2024].

[87] Merriam-Webster.com Dictionary, *Isomorphic*, https://www.merriam-webster.com/dictionary/isomorphic, [Accessed 16-05-2024].

[88] HexRays, *Ida sdk: Insn_t class reference*, https://www.hex-rays.com/products/ida/support/sdkdoc/classinsn__t.html, [Accessed 22-11-2023].

[89] cplusplus.com, *Function strcmp*, https://cplusplus.com/reference/cstring/strcmp/, [Accessed 25-03-2024].

[90] Q. C. Ltd., *Qt reference pages*, https://doc.qt.io/qt-6/reference-overview.html, [Accessed 25-03-2024].

[91] D. H. J. de St. Germain, *Functions*, https://users.cs.utah.edu/~germain/PPS/Topics/functions.html, [Accessed 25-03-2024].

[92] L. Baresi and M. Pezzè, 'An introduction to software testing,' *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 89–111, 2006, Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.12.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066106000442.

[93] A. Alhroob, K. Dahal and M. Hossain, 'Software test case generation from system models and specification. use of the uml diagrams and high level petri nets models for developing software test cases.,' Ph.D. dissertation, Dec. 2010.

[94] Damaye, Sebastien, *Peid*, https://www.aldeid.com/wiki/PEiD, [Accessed 27-03-2024].

[95] JetBrains, *C*, https://www.jetbrains.com/lp/devecosystem-2020/c/, [Accessed 10-05-2024].

[96] cppreference.com, *Std basic_string compare*, https://en.cppreference.com/w/cpp/string/basic_string/compare, [Accessed 03-05-2024].

[97] cppreference.com, *Atexit*, https://en.cppreference.com/w/c/program/atexit, [Accessed 03-05-2024].

[98] Python, *Timeit — measure execution time of small code snippets*, https://docs.python.org/3/library/timeit.html, [Accessed 03-05-2024].

Appendix A

GraphSlick's code

A.1 Updates done to GraphSlick

The section will describe the updates needed to make GraphSlick work with IDA Pro Version 8.3.230608 on Windows 10 x64. The updates are not meant to improve GraphSlick or expand its functionality. Instead, they are necessary as GraphSlick is around 10 years old, and a lot of its dependencies have changed in the last few years. The changes to GraphSlick have been documented in commits in a project on *github*. The following chapter will give a brief description of what the changes encompass.

When GraphSlick was developed Python 2 was the version shipped with IDA. This is no longer the case, and IDA uses Python 3. The IDA version used in this project uses Python 3.11.4. One of the main changes is to the string printing. As printing is handled as a function in Python 3, `print()`, this needs to be changed from how it was done in Python 2.

GraphSlick depends on the Python packages shown in Code listing A.1. All packages can be installed using the package installer for Python, pip. The packages are located at `pypi.org`. One of the packages, `ordered-set` needs to be modified to work properly. This change can be done by changing the file `orderd_set.py` on line 21 from `"class OrderedSet(collections.MutableSet)"` to `"class OrderedSet(collections.abc.MutableSet)"`. The file can be found at `%PYTHON_INSTALL_PATH%\Lib\site-packages\orderd_set.py`.

Code listing A.1: Python packages needed for GraphSlick

```
ordered-set==1.1
sark==8.2.0
pickleshare==0.7.5
```

The IDA API for Python has also changed since GraphSlick was developed. This is mostly related to the names of structures. An example of this is how the basic block object in IDA has changed the naming of its start and end address from `block.startEA` and `block.endEA` to `block.start_ea` and `block.end_ea`, and it has changed how operands are accessed in object instruction, from

`instr.Operands` to `instr.ops`. These changes to naming do not affect what properties are accessed. However, there would likely be some changes if an IDA version from 2014 was compared to version 8.3. The IDA analysis and disassembly process has improved in the last few years, and this will affect the properties accessed by the Python API as IDA's analysis creates the properties.

Appendix B

Test code

The appendix shows the code used when the test programs in Sections 6.2.1 and 6.2.2 were compiled. The source code for the program used in *Test 4* is also shown in this section as this was the same program as in *Test 1*. The command line used to compile the program or the compiler options are also listed in this appendix.

B.1 Test setup: *Test 1*

The test program contains the `std::string::compare`-function. The program uses a string taken from the command line, `argv`, as part of the string comparison to stop the compiler from being able to do the comparison compile time. This program was also used in *Test 4*.

The program were complied with cl, msbuild, and gcc, using the compilation arguments/command line shown below:

Code listing B.1: Comandline for compilation with clang

```
cl /EHsc /MD /O2 /Ob3 main.cpp
```

Code listing B.2: Options for compilation with msbuild

```
/permissive- /ifcOutput "x64\Release\" /GS /GL /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /
    Ob2 /sdl /Fd"x64\Release\vc142.pdb" /Zc:inline /fp:precise /D "NDEBUG" /D "
    _CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /Gd
    /Oi /MD /FC /Fa"x64\Release\" /EHsc /nologo /Fo"x64\Release\" /Fp"x64\Release\
    strcmp_msbuild.pch" /diagnostics:column
```

Code listing B.3: Comandline for compilation with gcc

```
g++ -O3 -MD -fno-exceptions main.cpp
```

Code listing B.4: Code example using strcmp

```
#include <string>
#include <iostream>

char ** ARGS;

int test_stringcmp(void) {
    std::string str = "";
    std::string sep = "␣&␣";
    for (char **it=ARGS; *it; it++) {
        std::string arg = *it;
        std::cout << arg << '\n';
        if (it!=ARGS) {
            str += sep;
        }
                else if (arg.compare(str) == 0 ){
                        break;
                }
        str += arg;
    }
    std::string cmpstr = "Test␣&␣string";
    int cmp = str.compare(cmpstr);
    if (cmp) {
                return cmp;
    }
        return str.compare(sep);
}

int main(int argc, char **argv) {
    ARGS = argv+1;
    return test_stringcmp();
}
```

B.2 Test setup: *Test 2*

Test 2 contained an inlined function from the same source code. Keywords were used to make the compiler inline a function. When the code was compiled with GCC the `__forceinline`-keyword was switched to `__attribute__((always_inline))`. The code was complied with cl, msbuild, and gcc with the compilation arguments/command line shown below:

Code listing B.5: Comandline for compilation with clang

```
cl /EHsc /MD /O2 /Ob3 main.cpp
```

Code listing B.6: Options for compilation with msbuild

```
/permissive- /ifcOutput "x64\Release\" /GS /GL /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /
    Ob2 /sdl /Fd"x64\Release\vc142.pdb" /Zc:inline /fp:precise /D "NDEBUG" /D "
    _CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /Gd
    /Oi /MT /FC /Fa"x64\Release\" /EHsc /nologo /Fo"x64\Release\" /Fp"x64\Release\
    Test_App_CFG_target.pch" /diagnostics:column
```

Code listing B.7: Comandline for compilation with gcc

```
g++ -O3 -MD -fno-exceptions main.cpp
```

Code listing B.8: Code example using inline-keyword in code

```
#include <Windows.h>
#include <iostream>

__declspec(noinline)
void func2(int i) {
        Sleep(i);
}

__forceinline
int func1(volatile int i) {
        int res = i;
        i++;
        if (i == 4) {
                res = res * 2;
                std::cout << "1" << std::endl;
        }
        else if (i == 2) {
                res = res - 3;
                std::cout << "2" << std::endl;
                func2(2);
        }
        else {
                for (int j = 0; j < i; j++) {
                        std::cout << "else" << std::endl;
                }
                func2(i);
        }

        if (res < 0) {
                return 0;
        }
        else{
                return res;
        }
}

int main(int argc, char* argv[])
{
        int temp;
        std::cout << "This␣is␣some␣code␣to␣add␣to␣the␣program" << std::endl;
        temp = argc ^ 0x1234;
        int res = func1(temp);

        if (res != 0) {
                func2(2);
        }
        else {
                std::cout << "Some␣more␣code␣here" << std::endl;
        }

        std::cout << "This␣is␣some␣more␣code" << std::endl;
        temp = func1(res);
        return temp + res;
}
```

Appendix C

Results full tables

This appendix contains the analysis results from the tests described in Chapter 6. The main results are described and visualized in Chapter 6. In this appendix the entire results are shown.

Appendices C.1 and C.2 contain tables of the inlined function detection done by the GraphSlick. The tables are split into two parts. The first part contains the results from the original GraphSlick, while the last contains the results from the modified GraphSlick.

Each row in the tables represents one group of detected isomorphic subgraphs. The first column is the group ID, and the following columns are the instances of block groups in that group. Thus, all block groups in one row are isomorphic subgraphs of one another. In each block group, the block IDs contained in that block group are listed. More block IDs in a block group mean that larger subgraphs were detected, whilst more block groups mean that more isomorphic subgraphs were detected.

Appendix C.3 contains tables of the CFG of the detected inlined function and the known library function. The tables list the block IDs and the corresponding *itype1_hash* for each block.

C.1 Results: Test 1: Detection of inlined string comparison

The results from *Test 1* are shown in this section. The results are contained in Tables C.1 to C.3. Larger block groups mean that more blocks were detected as part of an isomorphic function (potential inlined function).

Table C.1: Results of analysis done with original and modified GraphSlick on Code listing B.4 compiled with cl. The group ID is a hash of the blocks the group is comprised of, the block group contains the IDA block ID of the blocks in that block group. The results are described in Section 6.2.1.

Original GraphSlick

Group ID, SHA1	**Block group 1**	**Block group 2**	**Block group 3**	**Block group 4**	**Block group 5**	**Block group 6**
`ed7902b6b92af83b2eb35f6224a9f5b5245ddae6`	15, 16, 17, 18, 19	17, 18, 19, 20, 21	37, 38, 39, 40, 41	39, 40, 41, 42, 43	63, 64, 65, 66, 67	65, 66, 67, 68, 69
`1430c0f640fdee3d341469de0ca8caa2bfbac4d5`	36, 37	62, 63				
`6c21f2a1afbbe1be636298dd1e7aa50f4f9d4417`	27, 28, 29, 30	76, 77, 78, 79				
`6fa1ee4d434d04c90abafdc440dcb9d5c1580ffe`	11, 12, 13	33, 34, 35	59, 60, 61			

Modified GraphSlick

Group ID, SHA1	**Block group 1**	**Block group 2**	**Block group 3**	**Block group 4**	**Block group 5**	**Block group 6**
`c35227c5e4fdde80e28d35b1584bd20c98977aab`	10, 11, 14, 12, 15, 23, 13, 16, 22, 24, 17, 18, 19, 20, 21	32, 33, 36, 34, 37, 52, 35, 38, 44, 53, 39, 40, 41, 42, 43	58, 59, 62, 60, 63, 71, 61, 64, 70, 72, 65, 66, 67, 68, 69			
`f0c9585cc7119d8934dbb1bc6e905cf6181aac76`	76, 77, 78, 79	80, 81, 82, 84				

Table C.2: Results of analysis done with original and modified GraphSlick on Code listing B.4 compiled with msbuild. The group ID is a hash of the blocks the group is comprised of, the block group contains the IDA block ID of the blocks in that block group. The results are described in Section 6.2.1.

Original GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`e2505284662ad8d0688f 8c3de5306475c42aa2e3`	33, 34, 38, 35, 37, 39, 36, 40, 41	38, 39, 43, 40, 42, 44, 41, 45, 46
`11cadb4195b552541d83 4db22a59c440ea3baf50`	19, 20, 21, 22	43, 44, 45, 47
`b95e5d9c708ecc5efcca af86080f3d63d1e3dc5d`	24, 25	30, 31
Modified GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`11cadb4195b552541d83 4db22a59c440ea3baf50`	19, 20, 21, 22	43, 44, 45, 47
`b6e2913568685945b2ae a957b42ad9a61c1e1a32`	34, 35, 37, 36	39, 40, 42, 41

Table C.3: Results of analysis done with original and modified GraphSlick on Code listing B.4 compiled with GCC. The group ID is a hash of the blocks the group is comprised of, the block group contains the IDA block ID of the blocks in that block group. The results are described in Section 6.2.1.

Original GraphSlick			
Group ID, SHA1	**Block group 1**	**Block group 2**	**Block group 3**
`9df57c85bbbdde90c51b 1cfa0bf3ee7db2538913`	12, 13	14, 15	22, 23
`7ca2e83ffc3795c22881 105ef4c840c7c33ca1be`	41, 43	42, 44	
Original GraphSlick			
Group ID, SHA1	**Block group 1**	**Block group 2**	**Block group 3**
`9df57c85bbbdde90c51b 1cfa0bf3ee7db2538913`	12, 13	14, 15	22, 23
`7ca2e83ffc3795c22881 105ef4c840c7c33ca1be`	41, 43	42, 44	

C.2 Results: Test 2: Detection of inlined function using compiler keywords

This section shows the results from *Test 2*. The tables are in the same format as in Appendix C.1. The format is described in at the begining of the appendix.

Table C.4: Results of analysis done with original and modified GraphSlick on Code listing B.8 compiled with cl. The group ID is a hash of the blocks the group is comprised of, the block group contains the IDA block ID of the blocks in that block group. The results are described in Section 6.2.2.

Original GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`97c17ad28bc4fc23c042 1105406edfefa6f695c8`	2, 3, 4, 5, 7, 6	14, 15, 16, 17, 19, 18
Modified GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`14741e3274d04853bb45 52d2ea9fe4c4a4ca2f31`	0, 2, 3, 4, 5, 7, 6	12, 14, 15, 16, 17, 19, 18

Table C.5: Results of analysis done with original and modified GraphSlick on Code listing B.8 compiled with msbuild. The group ID is a hash of the blocks the group is comprised of, the block group contains the IDA block ID of the blocks in that block group. The results are described in Section 6.2.2.

Original GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`a29943fac0c91fc03a80 04ac3b2e72749ff2bd6b`	2, 3, 4, 5	11, 12, 13, 14
Modified GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`8632fb1a38b663651255 a7d52a80eea1af808398`	0, 1, 2, 3, 4, 5	9, 10, 11, 12, 13, 14

Table C.6: Results of analysis done with original and modified GraphSlick on Code listing B.8 compiled with GCC. The group ID is a hash of the blocks the group is comprised of, the block group contains the IDA block ID of the blocks in that block group. The results are described in Section 6.2.2.

Original GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`384fd4e1a43ee9c472b4 7664b37d4f0e82de0bb3`	1, 8	5, 9
Modified GraphSlick		
Group ID, SHA1	**Block group 1**	**Block group 2**
`928cdfba0280e6b2430a 72922a74e6762e3dbd1f`	0, 1, 10, 8	4, 5, 12, 9

C.3 Results: Test 4: Identification of string comparison function

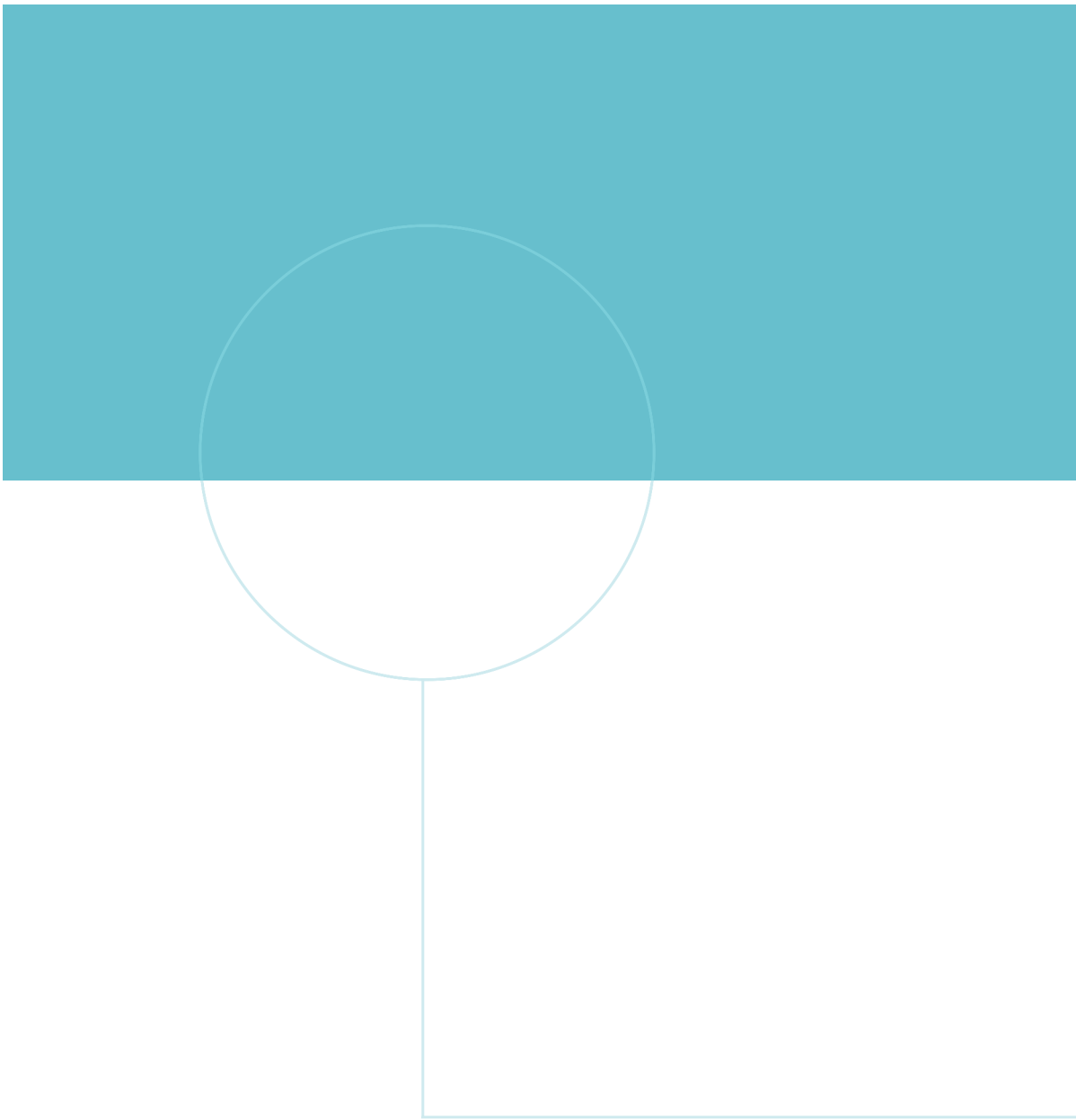
The CFGs from the detected inlined string compare function are listed in this section. The tables contain the block IDs and node hashes for the inlined function and the known string comparison function from the C++ standard library.

Table C.7: List of hash (itype1-hash) values of some of the blocks in the non-inlined version of the string comparison function compiled with cl.

Block ID	**Block hash**
0	`32ad14bd5184dee01d95c51a8d441dbc3192932d`
1	`2a0e2203060ee9b7af620be1d45db72dccf19e7e`
2	`958fe0b12065656a7e51b4131c3b6a8a0533572b`
3	`5bf3b6f6dce55a30d3e89c07bb1e1530b1e4eeeb`
4	`d52002b97a39cf545946dd413c3a1965b48abda8`
5	`8b7471f4ae0bf59f5f0a425068c05d96f4801b9e`
7	`eafcb28562dd561fba78f7c2aee929ec9495daf1`
6	`bad7183c5654a249eea49d487514850230733039`

Table C.8: List of hash (itype1-hash) values of the blocks contained in the inlined version of the string comparison function.

Block ID	Block hash
0	ae779acc189abad25fb53109993266c49ea21247
1	ae691977095934abd83cbc7b956ea10810bc2100
2	958fe0b12065656a7e51b4131c3b6a8a0533572b
9	c458d78de80be0321b9c18003f4a2327b1162f68
8	69e56976fc9bee70c1d2eaa85c0c8dea9f722a2f
3	6bade00c601dcdc1c1aeda947d467fdaa9fb3bf4
4	d548e8c90ebf6c25b931cfca81933a8bbe40dc09
5	8b7471f4ae0bf59f5f0a425068c05d96f4801b9e
7	b92265b94936e7d8282624aaf87e4ec89f601835
6	bad7183c5654a249eea49d487514850230733039
0xa	ca7469b2f65a9a05ad56851727e623528ab4e798
0xb	bfda6ee5117ee6b077bbc524328982c5d20abb03
0xc	ae779acc189abad25fb53109993266c49ea21247
0xd	1a16a42243784a2d0735add9326257d0fcbf07af
0xe	958fe0b12065656a7e51b4131c3b6a8a0533572b
0xf	6bade00c601dcdc1c1aeda947d467fdaa9fb3bf4
0x10	d548e8c90ebf6c25b931cfca81933a8bbe40dc09
0x14	1aaeb30f2b03b1508e5ab4c8e0257f013ad0ce7e
0x13	b92265b94936e7d8282624aaf87e4ec89f601835
0x11	8b7471f4ae0bf59f5f0a425068c05d96f4801b9e
0x12	bad7183c5654a249eea49d487514850230733039



NTNU

Norwegian University of
Science and Technology