MASTER'S THESIS

Switch Statement Disassembly

de Bruijn, M

Award date:
2022

SWITCH STATEMENT DISASSEMBLY

by

M.J. de Bruijn

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Day Month DD, YYYY at HH:00 PM.

Student number:

Course code:	IMA0002	
Thesis committee:	dr. Freek (F.) Verbeek (chairman),	Open University
	Prof. dr. Tanja (T.E.J.) Vos (supervisor),	Open University

CONTENTS

1

INTRODUCTION

Decompilation is a process of translating an executable binary file into a high(er) level language that, ideally, can be recompiled into an executable that preserves the semantics of the original binary. So it is the opposite of a compilation. A compiler compiles an executable file from a high(er) level source file. The decompilation process exists of multiple steps that lift machine instructions into a higher language or abstraction level. Each step recovers more information about the application so that eventually, a result in a higher language is created. The decompilation process starts with disassembly, lifts the binary into machine instructions, and is used for further processing. Unfortunately, even for the slightly easier decompilable languages like Java, the decompilation process is still far from perfect. Most decompilation tools produce a result containing both semantic and syntactic errors Harrand et al. [2019].

The software tasks where decompilation can be used are binary patching, porting, analysis, and improvement Verbeek et al. [2020]. Decompilation can be used because, in general, these tasks often assume that the source code is available. However, this is not always available due to intellectual property (IP) constraints or other issues. The task of binary patching is the process of repairing a known bug in the binary. Patching a binary without source code can be very difficult and complex Wang et al. [2017]. Binary Porting is the process of making a binary executable for a different processor architecture. The porting of binaries can be an alternative for emulators, which can slow down the application significantly. Binary analysis is used for, for example, software security or validation of safety-critical software. The low-level steps and interactions in binaries make analysis complicated and time-consuming. Binary improvement can make an executable execute faster due to better optimisation by a newer compiler version. When we recompile a binary with a newer compiler version, it can gain performance profits like a shorter execution time. So a *perfect* decompiler can make, in theory, all everyday software tasks possible even when the source code is initially not available.

Unfortunately, such a perfect decompiler does not exist. In the case of x86-64 binaries, disassembly is known as an undecidable problem. What causes this, for example, is the possibility of mixing data and instructions in the executable section of the binary Wartell et al. [2011]. This mixing of data and instructions is used for optimisation purposes, so a processor needs to perform fewer instructions to load the data into his registers. Indirect branching (computed jumps) and overlapping instructions can also cause improper

decompilation. Overlapping instructions is a technique that is frequently used by malware Meng and Miller [2016]. So the undecidability of some binary constructs in the x86-64 assembly can make those binaries hard to disassemble.

Disassembly is the first and an important step in the decompilation process. So the result of a decompiler depends heavily on the output of a disassembler. When a disassembler produces incorrect results, the decompiler will also do. Mainly two types of static disassembler algorithms exist, linear and recursive. The static recursive disassembly algorithm takes the application's control flow into account while disassembling. So it follows the code paths throughout the application. The linear algorithm does not disassemble all instructions of the binary consecutively. A decompiler will use the disassembly result and processes this into a higher programming language such as c.

As frequently researched, all decompilers produce different results for the same input Brumley et al. [2011] Brumley et al. [2013] Le et al. [2014] Meng and Miller [2016] Andriesse et al. [2016]. Each decompiler handles binary constructs or binary patterns produced by the disassembler in its way. Almost all decompilation papers are about common subjects like comparing a new decompiler or decompilation technique to existing decompilers. Those significant differences in decompiler results teach us that disassembly and decompilation are complex. Most tools perform well on some specific binary code, others do not, and vice versa.

The difference in results can primarily be assigned to the handling of corner cases or particular machine code constructs that exist in executable binaries Meng and Miller [2016] Andriesse et al. [2016]. However, the current state-of-the-art cannot precisely identify and handle all particular binary code constructs. Researchers who performed research on binaries are divided on the difficulties experienced during disassembly, decompilation or lifting. One other issue is that most studies on this topic only use a limited set of binaries and disassemblers. Therefore most of these studies can only make a statement about a small subset of binary executable binaries.

One of the things that are undecidable during disassembly is an indirect branch or indirect jump Verbeek et al. [2020] Andriesse et al. [2016]. For example, a programmer or compiler introduces an indirect branch while introducing function pointers, or a compiler implementation of a high-level programming language statement such as a `switch statement`. This indirect branching influences the control flow analysis of the disassembler. The control flow analysis of the disassembler has to determine which basic blocks the indirect branch can refer to. This process is challenging to figure out statically and can be very error-prone. Thus without the proper decompiler heuristics and algorithms, it is impossible to figure out the possible targets of the indirect branch.

A high-level programming statement frequently implemented by the compiler as an indirect branch is a `switch` statement. However, there are several different implementations a compiler can choose from. Some implementations use if-else constructs for a sequential test or binary search. Other implementations are based on lookup tables containing addresses or address offsets used in jump constructions. This jump-table construction creates an indirect branch. Some compilers place the jump-table, which is an array of data, in the executable section of the binary, which makes disassembly even more difficult Andriesse et al. [2016]. Also, the detected size of this jump table is essential. Missing entries can result in missing basic blocks. Extra entries can result in including not existing basic blocks. So the switch statement implemented by the compiler as jump-table implementa-

tion can make basic block recovery more challenging.

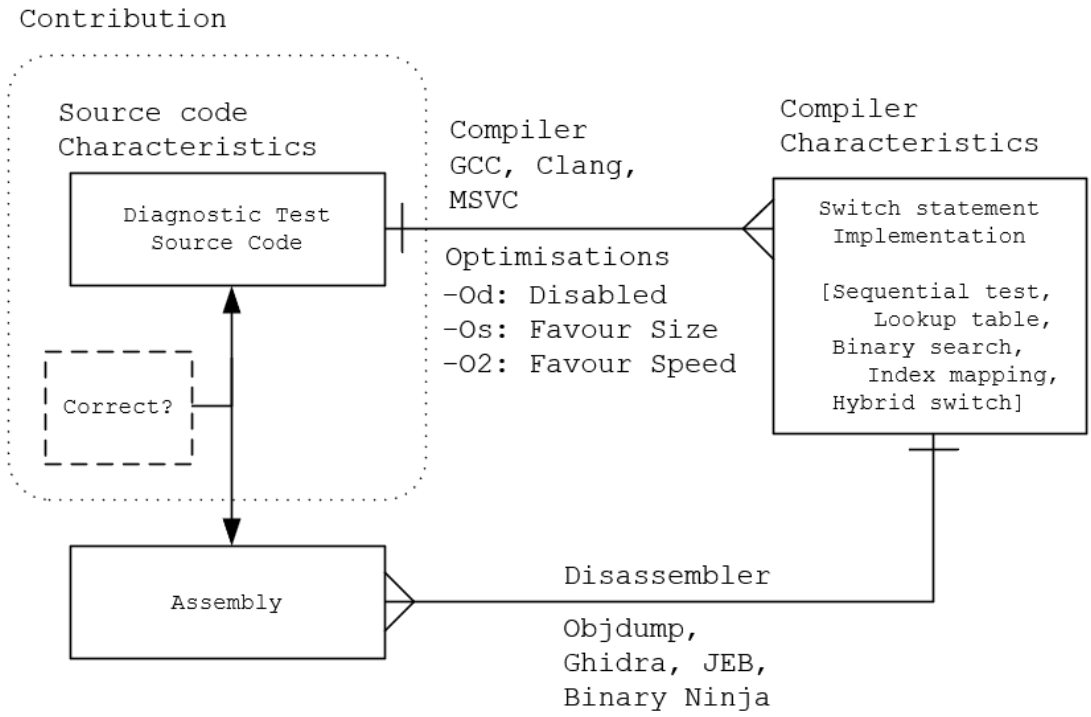

Figure 1.1: Contribution of this thesis in the dotted box

Most studies related to disassembly focus mainly on Linux OS and Linux executables and not so much on MS Windows. This while the Windows operating system is by far the most used desktop OS [1]. This lack of interest in Windows is probably because, for MS Windows, there is officially no source code available. Also, there is a greater variety of available compilers, which can produce different binaries and binary constructs.

In Figure 1.1 a schematic overview of the process used in this study is given. The contribution of this study is pictured in this figure as a dotted box. This contribution is the creation of diagnostic tests and verifying the switch statement disassembly results of several different disassemblers. The relation between the (intermediate) results, pictured as a solid box, is indicated as a one-to-many relation. The process starts with a diagnostic test that contains a piece of source code that contains a `switch` statement. The source code is fed to the mentioned compilers to create an executable binary containing a binary `switch` statement implementation. Hereafter the binary is disassembled to x86-64 assembly with the mentioned disassemblers. This resulting assembly code is used to verify the performance of the used disassembler regarding switch statement disassembly. In the last step, we will verify the assembly result for each possible combination of compiler and disassembler.

Our study aims to improve the current knowledge of available state-of-the-art disassembly tools regarding the decompilation of switch statements in Window executable binaries. To get insight into the performance of the different state-of-the-art disassemblers,

[1] https://gs.statcounter.com/os-market-share/desktop/worldwide

we will compare and verify the `switch` disassembly results. To verify the produced results, we will introduce some definitions later on. To be more precise, we perform the following actions consecutively. This study will:

- Define several `switch` statement diagnostic tests based on various source code and compiler depended characteristics
- Analyse the `switch` statement implementations of the compiled diagnostic tests for several compilers
- Verify the disassembly results of the compiled diagnostic tests of interest, disassembled with several state-of-the-art disassemblers
- Check if the found disassembly results are representative for found `switch` statement disassembly flaws in real-life MS Windows DLL binaries

Besides looking at the implementation of `switch` statements, we have also found another common issue for several disassemblers and we will explain this in chapter 4.

The result of our study shows that not all disassemblers used in our comparison can correctly disassemble all lookup table based `switch` statement implementations. Compilers have several options to implement a switch statement based on a `if-else` statement, a lookup table, or a combination of both. This results in different switch statement implementations depending on compiler and source code characteristics. The most important characteristics that influence the chosen implementation are the compiler itself, the used compiler optimisation setting, the number of case labels, and the used case value set. Because most of the analysed disassemblers use a static recursive disassembly algorithm, incompleteness of the produced `switch` statement disassembly result is the most frequently found issue. This issue arises because not all jump tables were interpreted correctly, like missing jump table entries or the entire jump table. Only one of the analysed disassemblers is a linear disassembler that produces unsound `switch` statement disassembly results for MSVC compiled binaries. This result is because the MSVC compiler places the jump tables between the instructions.

To the best of our knowledge, no other study has targeted the verification of various implementations of `switch` statements and the disassembly results hereof before. Also, the check on real-life Windows DLL executables gives us insight into the performance of the decompilers for real-world cases. We focus, like mentioned before, on switch statement implementations that use a jump-table implementation. So we will verify and compare the recovery of jump-tables, instructions and basic blocks. So the results of this type of analysis will give us more insight into the performance of disassemblers regarding the disassembly of `switch` statements.

Studies that relate to our study that we have found target the comparison of disassemblers Gusarovs [2018], Hamilton and Danicic [2009] Liu and Wang [2020] Harrand et al. [2019]. Our study differs from the mentioned studies because we focus on disassembly only. Another point from which we will differ is that for our real-life examples we use MS Windows DLL files, so we have no source code available. The studies mentioned as related work all use different types of input. Some use specially crafted input applications, and others use real-world, open-source applications. The handmade test cases contain complex code constructs like a loop with different entry and exit points that are randomly invoked or other complex code constructs. For comparison of the decompilation results, two

of the mentioned studies use equivalence modulo inputs testing Le et al. [2014] which was developed initially to validate compiler optimisations. One of the studies makes a manual comparison of the decompilation result, and the other uses software quality metrics as a comparison tool. The software metrics approach is a good method to measure the effectiveness of decompilers and obfuscators Naeem et al. [2007]. What also stands out for these studies is that most decompiler comparison studies target the Java programming language and only one other study targets decompilers for x86-64binaries.

The scope of our study is limited to `switch` statement validation and analysis. Also, we will only focus on the disassembly of binaries. We have based this choice on the fact that it is the first and most important input for the rest of the decompilation process. The targeted processor architecture for this study is x86-64. The x86-64 processor architecture is the most popular desktop and laptop PC architecture. The X86 architecture was introduced by Intel in 1978 and currently exists of a complex instruction set (CISC) that consists of more than 2000 different instructions.

The structure of the report is as follows. First, we focus on switch statement disassembly and the found issues. Therefore, we look at the possible switch statement assembly implementations, create diagnostic tests to test various switch implementations, validate the produced disassembly result, and look at real-life found disassembly flaws. Second, we will look at a different form of indirect branching issue related to pointer disassembly. This issue we encountered during our real-life disassembly analysis. Third, we discuss the studies that relate to this subject. Hereafter we give general information about disassembly and decompilation and introduce existing disassemblers and decompilers. Last, we have a discussion and conclusion about the study results.

2

BACKGROUND: DISASSEMBLERS AND DECOMPILERS

Decompilation and disassembly are fields that are already studied for several decades, starting somewhere in the '60s. On the internet, there is a comprehensive overview of the history of decompilers [1].

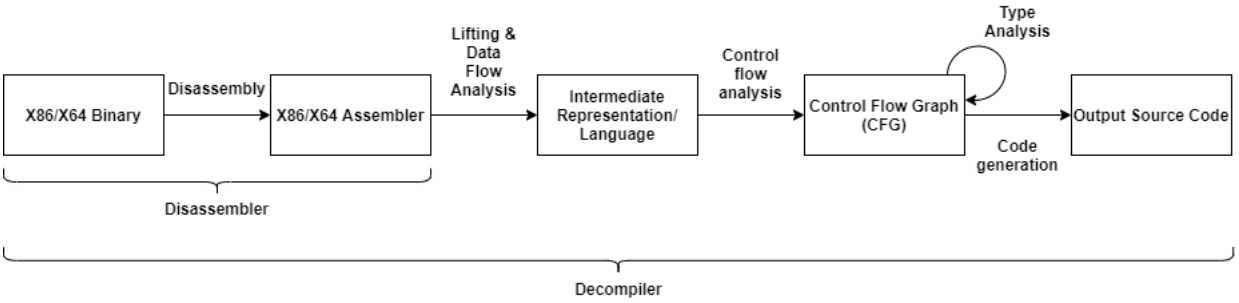

Figure 2.1: A general overview of disassembler and decompiler processing steps with their relations.

Decompilation and disassembly is a process that can be seen as a chain of linked smaller processes, and each has different inputs and outputs. The difference between a disassembler and a decompiler is that a disassembler converts the binary into human-readable machine instructions, and a decompiler converts the binary into a high(er) level programming language. Disassembly is also an essential process step of the decompilation process. The decompilation process exists of more steps than a disassembler and is more complex than disassembly. Disassembly and most of the decompilation steps are still part of ongoing academic research, so there is no general consensus about how they should be performed. A general visual representation of the decompilation and disassembly steps is shown in Figure 2.1. Not all decompilers perform necessarily the steps in the shown order or use all processes and outputs.

Other essential steps of the decompiler process are the variable/type analysis, control-flow recovery and the optimisation steps during the code generation. The steps are essential because these steps recover the variables, data types and high-level control structures.

[1] http://www.program-transformation.org/Transform/DeCompilation.html

In other words, decompilation recovers all kinds of information that is disregarded by the compiler and is needed to process the binary into a high-level language.

Most decompilers produce results with understandability for the human reader in mind. They produce results that are of a higher abstraction level and focus on a better readability and code size than the binary input. This result is no longer sufficient for the emerging automated software analysis, testing, and verification fields. We want the decompiler to produce sound code and compilable code instead of readable code. This transformation into sound code also applies to legacy software maintenance tasks where no source code is available. Therefore the more recent studies target the soundness and recompilability of the produced results Brumley et al. [2013]. One other recent study takes this even further. They even provide a formal verification for the soundness of the produced results Verbeek et al. [2020]. They do this for an x86-64 binaries subset, so this method has its limitations.

Tool	x86-64 Input	Output	IL	Types	API
BAP	ELF & PE	-	BIL	dynamic or static	✓
FoxDec	ELF	C	Abstract Code	type punning	
Ghidra	ELF & PE	C	P-code	static	✓
Phoenix	ELF & PE	C	BIL	dynamic or static	✓
Binary Ninja	ELF & PE	psuedo C	BNIL	static	✓
JEB	ELF & PE	C	JEB IR	static	✓
Objdump	ELF & PE	ASM	-	static	

Table 2.1: Enummeration of x86-64 disassembler/decompilation tools.

Table 2.1 includes several recent disassemblers and decompilers. They are compared to their type of input, output, Intermediate language (IL), type recovery system, supported architectures, and availability of an API for user extensions. The mentioned tools are all binary decompilers, so they process binary executable files into an output which often is C code. The input indicates if the tool handles the following x86-64 binary files: Linux type format (ELF) or the Windows type format (PE).

Almost all tools have a different intermediate language (IL) implemented. The translation of the binary into an intermediate language is called *binary lifting*. After lifting the binary, it is processed to become the required output eventually. The abstraction level of the IL lies between the assembly code and the output code (in a higher programming language mainly) and differs for each implementation. So the level of abstraction must be wisely chosen because of the loss of binary information. Some decompilation tools like Binary Ninja have multiple ILs, which all have a different abstraction level [2]. The use of an IL makes it easier to expand the decompiler to different processor architectures. The IL is more effective for further analysis and processing than assembly code. The input binary code is disassembled and lifted into the IL and represents a control flow graph of the application (CFG). This CFG exists of vertices representing instructions or code blocks, and the edges represent the code paths throughout the application.

[2]https://docs.binary.Ninja/dev/bnil-llil.html

The recovery of data types is challenging because the compiler throws this high-level type of information away. We have mainly two options to recover the variables and their data types; the first and most frequently used for the previously mentioned decompilers is *type inference*. The second option is *type punning*.

Type inference uses a type reconstruction algorithm that finds type constraints for each instruction (the use of the variable) to deduce the data type of a variable Mycroft [1999]. This type of recovery algorithm can be used dynamically and statically. The dynamic approach uses the control flow while running the application. During the execution of the application, it analyses the use of the variables. With this information, it tries to deduce the data type. The static approach analyses the code and considers well-known function signatures to infer the data types. Both type inference approaches have pitfalls, the dynamic approach has a lower program coverage, and the static approach can use inaccurate heuristics. Both approaches for type inference can lead to wrong type assumptions.

Type punning is a method to assume a default data type and cast a variable to the appropriate data type for their use. Type punning is a far more easy approach than type inference. This approach is much easier because typecasting on the spot needs fewer things to take into account. Therefore this approach is more likely to result in correct and recompilable code. So type punning delivers recompilable code more frequently with the downside of being less human-readable. Also, type inference is not always possible due to undecidability.

Binary Analysis Platform (BAP) is an open-source binary analysis and verification tool from the Carnegie Mellon University Brumley et al. [2011]. The application and source code is publicly available. BAP consists of a collection of tools for performing program analysis and verification of binary code. BAP is developed and actively maintained and supported by Carnegie Mellon University. The application itself is a further development of previously developed academic tools for binary analysis. The architecture of BAP consists of a front-end and a back-end. The front-end lifts the binary to an IL, and the back-end performs analysis or program verification. As an intermediate language, BAP uses a self-defined language called BAP Intermediate Language (BIL). The advantage of BIL is that it makes all side effects of the assembly code explicit. These side effects arise in the binary code because some x86-64 instructions also depend on the processor status flags. BAP uses a linear sweep algorithm to process the x86-64 binary machine code. BAP uses a type recovery system called TIE Lee et al. [2011]. This type of inference system can be used for both static and dynamic type inference and analyses memory access to find the variable locations and analyses their usage to determine their type. Each usage of a variable imposes some constraints on the variable's data type. TIE is more accurate in finding data types than HexRays and the other academic developed REWARDS dynamic type inference system. Because BAP is an analysis tool, it produces no source code by itself; however, extensions through an API can be created that can produce source code. BAP has a well-documented API which makes it possible to create libraries, plugins and front-ends [3]. The limitations of BAP are that it cannot handle floating-point datatypes and it can not handle certain exotic x86-64 instructions Brumley et al. [2013].

FoxDec (Formal X86-64 Decompilation) is a sound and recompilable C Code decompiler Verbeek et al. [2020]. The most crucial difference between FoxDec and onther de-

[3] http://binaryanalysisplatform.github.io/bap/api/odoc/index.html

compilers is that the processing is formally verified for a subset of x86 binaries. This formal verification is performed for three key stages of the decompilation process. Because of this verification, the tool delivers sound and recompilable code. Sound in this context means that the extracted C code behaves exactly the same as the original binary. FoxDec has a special developed IL datatype called *abstract code*. This datatype can express control flow, basic blocks, and branching decisions. The difference between the previously mentioned ILs and abstract code is that formal proof is provided for the soundness of the IL. This decompiler is the only decompiler in this list that uses type punning as a type recovery method. This approach is beneficial for the recompilability but makes the C code less human-readable. FoxDec has, besides great benefits, some limitations in usability. Those limitations are that human interaction is needed for function signatures, the inclusion of header files and FoxDec can not deal with indirect branching.

Recently, the US National Security Agency (NSA) has released an opensource version of their decompiler *Ghidra* [4]. This decompiler targets the analysis of malicious code. This decompiler does not necessarily produce sound and recompilable code. *P-code* is Ghidra's intermediate language. P-code is a simplified instruction set that is developed to make the analysis of binary code easier [5]. The IL is generic enough to be able to model the behaviour of many different processors. Because of the design of this IL, Ghidra processes a machine instruction into a sequence of multiple P-code instructions. As a type recovery method, Ghidra uses a static type inference approach that uses a register-based data-flow analysis Zhang et al.. Ghidra can easily be extended by making use of a well-documented API.

The *Phoenix* decompiler is, like BAP, developed by the Carnegie melon university and also targets the soundness of the decompilation results Brumley et al. [2013]. Phoenix is an extension of BAP with security analysis in mind. The difference between FoxDec and Phoenix is that no definition for soundness is given. Verification of Phoenix is done by decompiling a tools set, recompiling them and running a test suite. Hereafter the result is compared with the de facto decompiler HexRays, which for the applied test cases, Phoenix outperforms. Phoenix has the same limitations as BAP because of the shared modules. Because Phoenix uses BAP, it also uses the same IL type recovery method (TIE), and it is also extendable by creating plugins for BAP.

Binary Ninja [6] is a commercial reverse-engineering platform that is developed for binary analysis. It was developed by Vector 35, and its release was in 2016. So it is not that long available for the public. Before its public release, it was an internal tool for a hacker group that used it for CTF tournaments. Because it targets binary analysis, it will produce various views of the binary, such as a control flow graph and several intermediate textual representations. Binary Ninja uses several intermediate languages, each with a different abstraction level. All can be easily viewed and analysed in the user interface. It uses a static recursive disassembly algorithm. API and plugin options are available that make it possible to create or extend functionalities, analysers, and other processing modules.

JEB is created and maintained by Pnfsoftware and supports a variety of processor architectures. Pnfsoftware was founded in 2013. JEB is a commercial reverse engineering tool known for android decompilation. It supports dynamic and static analysis for multiple architectures and executable formats, including x86-64 PE and ELF binaries. For disassembly,

[4] https://github.com/NationalSecurityAgency/ghidra
[5] https://ghidra.re/courses/languages/html/pcoderef.html
[6] https://binary.ninja/

it uses a static recursive algorithm. JEB also supports a well-documented API, plugins and extensions.

Objdump is part of the open-source GNU Binutils collection. This collection contains a collection of binary tools for Linux. Objdump is the only tool in this list that does not provide an API and uses a linear disassembly algorithm. Objdump uses the Binary File Descriptor library for low-level file information and the opcodes library for disassembly. The tool can disassemble both PE and ELF files compiled for the x86-64 architecture.

3

SWITCH STATEMENT DISASSEMBLY

A `switch` statement provides a multi-way branch in the higher-level programming languages such as C/C++ and Java. This statement is used to manipulate the control flow in an application. A `switch` statement is a flexible way to execute different code blocks depending on an expression. This statement can be seen as multiple `if` statements or a `goto` statement. Depending on the value of the `switch` expression, the program jumps to a symbolic label. So with the `switch` statement, the application can take one of the multiple branches depending on the switch expression value.

This chapter will first discuss typical compiler `switch` statement implementations and how compilers implement these at the machine code level. Hereafter we introduce the definitions that we will use to verify the switch statement implementations. Third, we will verify how compilers implement `switch` statements by analysing diagnostic tests. We will also use these tests to compare the disassembly results of multiple disassembly tools. At last, some real-life switch statement errors are discussed.

3.1. SWITCH STATEMENT ASSEMBLY IMPLEMENTATIONS

A compiler has several options to implement a `switch` statement in assembly Sayle [2008]. The most common are; the unconditional branch, sequential tests, jump table, binary search, and index mapping.

A compiler will use the *unconditional branch* only when the default label statement is given or when the default label statement is the only one left. This branch will result in a `JMP` instruction to the default label instruction(s).

Sequential test is the most universal implementation. Each case label statement is translated into an individual if-else statement. The application sequentially checks all if-else statements until the case value is found.

A `switch` statement can also be implemented as an *indirect jump*. An indirect jump is a jump whose target address is computed at run-time. This type of implementation uses the switch value to index an array of addresses or address offsets, and it uses the address to jump to directly. The offset will be used in a calculation that results in the desired address. To reduce the size of the array, normalisation of the switch value need to be performed. The normalisation will let the range of switch values begin at zero for the first case.

When the values of case label statements are more sparse, a solution based on the *binary search* algorithm is chosen. This balanced search tree starts with a check on the median value of the case label statements. Hereafter the lower or higher branch is chosen. The implementation of this algorithm will reduce the search time to *O(log n)* instead of *O(n)*.

Some compilers like MS Visual Studio also use a method that is called *index mapping*. This method is also a table based technique like jump tables with broader applicability. The difference is that this method uses two arrays (tables) instead of one. The extra lookup table is used to find the address index. The advantage of this approach is that it forms the address table into a dense zero-based indexed array. Another advantage is that the range of the first table (the address index table) is known, so a range check on the address table itself can be omitted.

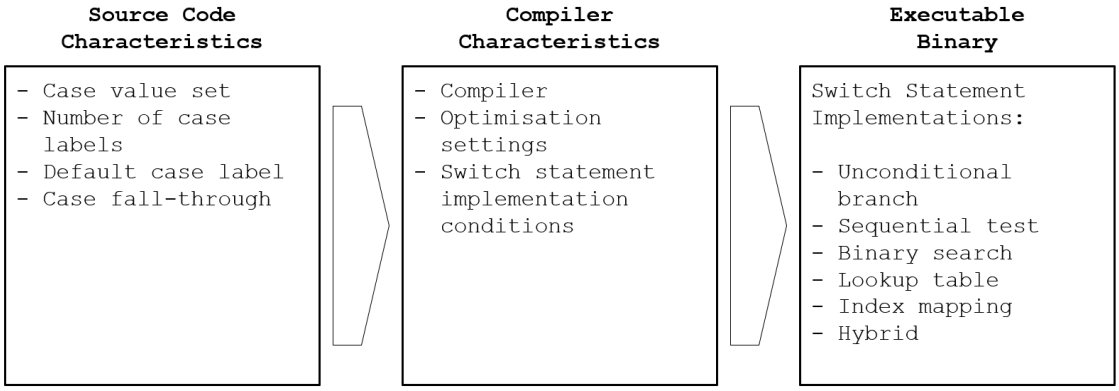

Figure 3.1: Switch statement implementation characteristics.

A compiler's approach to implementing a case statement depends on several characteristics. These characteristics are source code and compiler dependent. So which `switch` statement implementation a compiler chooses depends on the switch statement source code and compiler characteristics and settings, such as the optimisation settings. It is even possible that a compiler chooses a hybrid variant and divides a `switch` statement into multiple `switch` implementations mentioned above. For example, when two spaced ranges of case label expressions are used.

In Figure 3.1 is given a schematic overview of creating a `switch` statement implementation in the executable binary. This picture represents both the characteristics and the compilation process and their relation. For the process and each of the characteristics, an explanation is given below. The most frequent used `switch` statement are given in the executable binary representation.

This process is the general process of compiling a binary. First, a piece of source code is created which contains the source code representation of a `switch` statement implementation. The source code is used as an input for a compiler which will, during compile time, compile the source code into an executable binary. The result of this process is an executable binary that will contain a binary implementation of the in source code given `switch` statement.

The most influential source code characteristic for creating a `switch` statement implementation is the number of case labels and the case value set. The number of values and their interval is relevant for the case value set. Other less influential source code characteristics are if a default case label or a case fall-through is being used. A case fall-through is created when no `break` statement is applied for one or more case labels. This will result in the execution of different case labels until a `break` statement is found.

The optimisation setting is a user parameter and instructs the compiler to make choices that optimise the resulting binary. For a compiler, relevant characteristics are the compiler itself, the optimisation settings and the `switch` statement implementation conditions. The implementation a compiler chooses is dependent on the predefined or known `switch` statement implementations by the compiler and the related implementation conditions. The optimisation setting will finally optimise the chosen implementation in the manner desired by the user. So if a user wants a faster execution time, the user can instruct the compiler to make choices that favour speed.

For example, when only a few case expressions are provided, most compilers will be inclined to implement the `switch` statement as a *sequential test*. Thus a list of `if` statements. When more case label expressions are given, and their values are dense, a compiler will choose the *lookup table* variant. When the values of the case statements are more sparse, a compiler prefers to implement a *binary search*. Other source code characteristics that will influence the implementation are when a default case label or case fall-through is applied.

As an example a simple `switch` statement is given in Listing 3.1. It consists of a `switch` statement and as case blocks some assignments. The following `switch` statement source code characteristics can be found:

- Case value set {0, 1, 2, 3}
- Number of case labels: 5
- A default case label
- A case fall-trough from case label value 2 to case label value 3 (no `break` statement applies)

The decision a compiler makes regarding which type of `switch` implementation depends on the compiler characteristics. These characteristics relate to the compiler, switch statement implementation conditions, and optimisation setting. Those conditions will guide the compiler in choosing an implementation. For example, a condition, or a threshold, is the number of case labels. So when only a few case values are used, less than this threshold, a compiler uses a sequential test. The compiler optimisation setting also influences those conditions and guidance. So an optimisation level favour speed (`-O2`) will result in faster implementation, like index mapping, and an optimisation setting `-Os` will result in an implementation that requires less code size.

Another compiler dependent characteristic is the location of the placing of the jump table itself. When the visual studio compiler decides to use a jump table, the values of the jump table are placed right after the end of the function. This placing is in contrast to GCC, Clang and other C compilers, which place the jump table into the read-only data section Andriesse et al. [2016]. This mixing of data and instructions will make an executable harder

```
switch (input) {
case 0:
        output = "Case 0";
        break;
case 1:
        output = "Case 1";
        break;
case 2:
        output = "Case 2";
case 3:
        output = "Case 3";
        break;
default:
        printf("Err. Invalid input!\n");
        exit(1);
}
```

Listing 3.1: A simple `switch` statement example.

to decompile, resulting in data being treated as code and thus missing basic blocks.

3.2. SOUNDNESS AND COMPLETENESS OF DISASSEMBLY

We will only look at `switch` statements based on table-based solutions such as jump tables and index mapping to verify the decompilation process of case statement disassembly. We base this decision on the fact that other case statement implementations, such as a sequential test and the binary search, will result in if-else statements that are easier to decompile because these constructs are more straightforward and more prevalent in the code.

First, we will introduce the definition *reachable*. In our context, an instruction address is reachable, if and only if, a control flow path exists from the entry point of the executable binary to that instruction address. A basic block is reachable if and only if its first instruction is reachable. These control flow paths can be determined for both static and dynamic analysis. Static analysis in this context means that the analysis takes place without running the examined binary. The dynamic analysis takes place while running this binary. We will only focus on static control flow paths during our analysis of the binaries. So we will only analyse the control flow paths that we will find by means of static analysis.

The resulting disassembled `switch` statements and the found control flow paths we are going to analyse these on both *soundness* and *completeness*.

Definition 3.2.1 (soundness)**.** For the `switch` statement analysis, we define *soundness* as follows: If the disassembler produces a case basic block, then this case basic block is also reachable in the executable binary.

Definition 3.2.2 (completeness)**.** The definition we use for *completeness* is as follows: If a case basic block is reachable in the executable binary, then this case basic block is produced by the disassembler.

In Listing 3.2 an example hereof is given. So let `rax` be a value between 0 and 3. The

disassembler is sound if it produces instruction i_0 till i_3. If the disassembler produces instruction i_4, the result is unsound because this instruction is unreachable in the input binary. So in the case of `switch` statement disassembly, the result of a disassembler could be bound to table size and table values. A disassembler is *complete* if it produces instructions i_0 till i_4. The result is incomplete when a disassembler misses one of the instructions i_0 ...i_3.

So the result of a linear sweep disassembly algorithm is often complete, it does not miss any instructions, but unsound, it produces unreachable instructions. The result of a recursive disassembly algorithm is often sound, it produces only reachable instructions, but incomplete, it does not disassemble all instructions because of indirections in the binary.

```
mov rax, {0,1,2,3}
jmp a + 8*rax

a+0:
    i_0
a+8:
    i_1
a+16:
    i_2
a+24:
    i_3
a+32:
    i_4
```

Listing 3.2: switch statement pseudo assembly

Note that we have left out the case label conditions in the provided definitions for soundness and completeness. This is because we consider static reachability only. We chose this because it creates a more generic definition but is still specific enough to perform our research. So, despite having left out the case label conditions, our definitions will give us enough information and insight to understand the `switch` statement disassembly ability of the to be tested disassemblers.

3.3. SWITCH STATEMENT DISASSEMBLY

In this section, we will analyse how some compilers implement `switch` statements and how disassemblers can disassemble those `switch` statements. First, we will create some switch statement diagnostic tests. Herefore we have defined characteristics of interest and translated them into a diagnostic test, see Figure 3.1. Second, we have compiled the diagnostic tests with several compilers and compiler settings and analysed the `switch` statement implementations. At last, we disassembled the diagnostic tests with several disassembly tools and verified the produced results.

3.3.1. SWITCH STATEMENT DIAGNOSTIC TESTS

We have developed several diagnostic tests for the `switch` statement disassembler analysis. An overview of the diagnostic tests and their characteristics can be found in Table 3.1. In this table, we have summarised the characteristics of each of the created diagnostic tests. Each diagnostic test has a reference label which will be used throughout the rest of this chapter.

Diagnostic Test	value set	No. case labels	default label	enclosed if statement	nested switch statement	enum case constants	case fall-through	function pointers	nested loop
C1	Consecutive	3	✓						
C2	Consecutive	10	✓						
C3	Consecutive	10		✓	✓	✓	✓	✓	✓
C4	Consecutive	100	✓						
R1	Ranged	3	✓						
R2	Ranged	10	✓						
R3	Ranged	10		✓	✓	✓	✓	✓	✓
R4	Ranged	100	✓						
S1	Sparse	3	✓						
S2	Sparse	10	✓						
S3	Sparse	10		✓	✓	✓	✓	✓	✓
S4	Sparse	100	✓						

Table 3.1: Switch table diagnostic tests.

As can be seen in this table, there are mainly three types of diagnostic tests; each type has a different switch value range and interval. The ranges of diagnostic test values we classify as *consecutive*, *ranged* and *sparse*. The values used for the tests are all natural numbers including zero. Examples of the case values sets are:

- Consecutive = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Ranged = {0, 1, 2, 3, 4, 500, 501, 502, 503, 504}
- Sparse = {3, 6, 7, 10, 13, 16, 19, 22, 25, 29}

The *minimal diagnostic tests* provides a minimal code example of a `switch` statement. The tests are C1, C2, R1, R2, S1, and S2. Depending on the switch value, it sets a string value that is printed on the screen right after the `switch` statement. The switch value is a parsed command-line value string to an integer. With these diagnostic tests, we want to test if a disassembler can disassemble all data and instructions related to `switch` statement.

The core of the *extensive diagnostic test* is a `switch` statement with ten case values. The extensive test are C3, R3, and S3. In this type of diagnostic test the `switch` statement is dressed with more code constructs in and around the `switch` statement to make the disassembly process more complex. We have chosen to use code constructs that you will encounter also in real-life `switch` statements. These code constructs are, for example, an `enum` for `case` label values, function pointers, nested `switch` statements and `if` statements , case fall-through and loops. With this diagnostic test, we want to test the influence of these different code constructs on the disassembler and detect the `switch` statement and the extra code constructs.

For each value set, we have also created an *extra long diagnostic test*. The references for these tests are C4, R4, and S4. The structure of these tests is the same `switch` statement construction as for the minimal diagnostic tests. However, for this diagnostic test type, we have extended the `switch` statement up to one hundred case labels and a default case.

3.3.2. COMPILER SWITCH STATEMENT IMPLEMENTATIONS

To analyse what type of diagnostic test is of interest for our test, we will first analyse the type of `switch` statement code construct a compiler creates. This analysis is performed for several switch statement diagnostic tests. Each compiler uses different heuristics regarding translating the high-level programming language into machine instructions. So using different compilers for the same input, the machine instruction output would not be alike, although the software functionality is.

The compilers we will use for our comparison are the GNU Compiler Collection (GCC), Microsoft Visual C++ Compiler (MSVC) and Clang. GCC [1] is one of the most popular compilers for compiling C code. GCC is an open-source compiler and can compile many different higher-level programming languages. This compiler is maintained by GNU and is part of the GNU programming tool-set for the Linux kernel. These tools are also available for the Windows OS with the Cygwin tool-set. MSVC is the Microsoft C compiler provided with Microsoft Visual Studio and is the compiler used by MS to compile DLL files and other kernel-related modules. Clang is a compiler for C, C++, Objective C and Objective C++ programming languages and acts as a drop-in replacement for GCC. Clang is open-source and based on the LLVM[2] Compiler infrastructure project and supports several architectures. LLVM provides a language-independent intermediate representation and optimisations and is used by multiple compilers.

The used versions of each decompiler are:

- GCC, Version 10.2.0
- MSVC, Version MS Visual C++ 2019 - 00435-60000-00000-AA67
- Clang, Version 10.0.0

The diagnostic tests we will use are the ones that are mentioned in Table 3.1. We compile each diagnostic test with several optimisation settings and analyse the impact of the `switch` implementation on the produced output. The optimisation settings we will use for our analysis are: `-Od/-O0` (optimisations disabled) `-Os` (favour application size) and `-O2` (favour application speed).

For each diagnostic test and compiler setting, we have analysed the implemented `switch` statement approach of the compiler. The result of this compiler test is noted in Table 3.2. The rows of this table represent the diagnostic tests, and the columns represent the compiler and the compiler settings.

As can be seen in the results (Table 3.2), the compilers create different types of `switch` statement solutions depending on diagnostic test and compiler settings. For the diagnostic tests with 3 case labels, mostly a *sequential test* implementation for `switch` statements is chosen. An exception hereof is diagnostic test C1 compiled with Clang, and Clang chooses to implement a lookup table for the optimisations `-Os` and `-O2`. A table-based lookup

[1] https://gcc.gnu.org/
[2] https://llvm.org/

Diag.	GCC			MSVC			Clang		
Test	-Od	-Os	-O2	-Od	-Os	-O2	-Od	-Os	-O2
C1	1	1	1	1	1	1	1	2	2
C2	2	2	2	2	3	2	2	2	2
C3	2	2	2	2	3	2	2	2	2
C4	2	2	2	2	3	2	2	2	2
R1	3	1	1	1	1	1	1	1	1
R2	5(2,2)	5(2,2)	5(2,2)	1	1	1	1	5(2,2)	5(2,2)
R3	5(2,1)	5(2,1)	5(2,1)	1	1	1	1	5(2,1)	5(2,1)
R4	5(2,2, 2,2)	5(2,2, 2,2)	5(2,2, 2,2)	5(2,2, 2,2)	5(3,2, 2,2)	5(2,2, 2,2)	1	5(2,2, 2,2)	5(2,2, 2,2)
S1	1	1	1	1	1	1	1	1	1
S2	2	2	2	2	3	2	2	2	2
S3	2	2	2	4	3	4	2	5(2,1)	2
S4	2	3	2	5(4,4)	3	5(4,4)	2	3	2

Table 3.2: Switch table implementations per compiler.

1 = sequential test
2 = lookup table
3 = binary search
4 = Index mapping
5 = hybrid switch (made up of ..)

variant is mostly chosen for diagnostic tests with more case labels.

For the *consecutive* diagnostic tests with a higher switch count (>3), the lookup table is the most prevalent used solution by all three compilers. Only MSVC chooses for the optimisation -Os to implement a binary search construction instead of a lookup table based solution. Clang takes the optimisation of the diagnostic test C2 and C4 for optimisation settings `-Os` and `-Od` even further than MSVC and GCC. It creates a lookup table for the strings, passed to the print function as a pointer.

The *ranged* diagnostic tests show a more distinct result between the three compilers for the diagnostic tests with a higher case label count. GCC and Clang cut the `switch` statement in half and treat them separately by doing a range check for the upper and lower value range prior to the `switch` statement. The diagnostic test R2 is divided into two lookup table-based solutions. The diagnostic test R3 is divided into a lookup table solution for the lower range and a sequential test for the upper range. They probably choose a sequential test for the upper range instead of a lookup table due to the fall-through of case labels 5, 6 and 7. This fall-through results in three basic blocks for the upper range and, like the other `switch` statements with a lower case label (or code block) count, this will result in a sequential test. One exception is the ranged diagnostic tests compiled with Clang and no optimisation; these are implemented as a sequential test. This choice is made most probably for the sake of simplicity. MSVC implements all ranged diagnostic tests as two sequential tests for the different range values and a range check before. However, the ranges for ranged tests R2, and R3 of the MSVC implementation are not precisely divided

into the higher and lower range like GCC. For example, the value of case 5 (value 500) is also included in the lower range.

For the implementation of the extra-long, ranged diagnostic test R4, containing a switch statement with 100 case labels, it is noticeable that in almost all cases, the same kind of hybrid solution was chosen. The only exception is the test compiled with Clang `-Od`. All switch statements are divided into four segregated sub-switch statements with extra range checks for each sub-switch for this hybrid implementation. The case label values are all equally divided into 25 case values. For the sub `switch` statements, GCC and Clang produce, in all cases, a lookup table. MSVC produces as an exception for the optimisation setting `-Os` one binary search for the lowest range of 25 case values. So the choice for this hybrid implementation is independent of the compiler and optimisation setting. one minor difference in implementation is that MSVC for the optimisation setting `-O2`, for three of lower the sub `switch` statements, chooses to implement one separate case label as an `if` statement. The case label value chosen for this `if` statement the highest value of the sub `switch` statement.

The *sparse* diagnostic tests show us also some diversity between the three compilers. GCC chooses to implement a lookup table for the large and extended version independent of the compiler optimisation setting. MSVC shows a more diverse `switch` statement solution among the cases and optimisation setting. This compiler prefers a lookup table solution for S2 and an index mapping for S3 diagnostic test. Like the consecutive diagnostic tests, MSVC chooses to implement the `switch` statement as a binary search for the favour size optimisation setting (`-Os`). Clang's implementation of the `switch` statement is almost entirely similar to GCC. One exception is S3 and optimisation `-Os`, this `switch` statement is implemented as a hybrid `switch` statement just like R3. The compiler's choice for implementing the extra-large sparse diagnostic test S4 does not differ a lot mutually and from the preceding tests. The difference that can be noted is that MSVC creates a hybrid of an index mapping implementation for the optimisation `-Od` and `-O2`.

So MSVC uses a greater variety of switch statement implementations than GCC and Clang. This compiler is also the only compiler that uses an index mapping implementation for our diagnostic tests. The optimisation setting has a greater impact on the chosen implementation strategy than the other compilers. The optimisation setting `-Os` will more frequently result in a sequential test or binary search type solution. This solution is, for our tests, less space-consuming than a lookup table based solution. For the other optimisation settings (`-Od -O2`), a lookup table based solution is chosen.

GCC chose most often for a hybrid `switch` solution in comparison with the other compilers. It has implemented most of the ranged diagnostic tests as a hybrid `switch` statement with equally divided ranges. GCC's switch statement solution differs only in minor details for the different optimisation settings. This is in contrast, as mentioned before, to MSVC. GCC is more likely to choose a lookup table as the preferred solution for large or extended diagnostic tests and a sequential test for the lower basic block counts.

In general, Clang switch statement implementations look a lot like GCC. However, for some optimisation settings, a different solution is chosen. Nevertheless, the set of implementations from which is chosen is equivalent.

3.4. DISASSEMBLY OF DIAGNOSTIC TESTS

We will analyse the disassembly results for the diagnostic tests that use a lookup table or index mapping as switch statement implementation. The used switch statement implementations can be found in Table 3.1. This comparison is performed for several disassemblers/decompilers. The disassembly result we will verify on soundness and completeness as defined in Section 3.2. As input for the disassembler, we will use the compiled diagnostic tests from Section 3.3.2.

For the diagnostic test analysis we will use the following four disassemblers:

1. Ghidra, Version 10.1 PUBLIC
2. JEB, Version JEB 4.14.0.202203082008
3. Binary Ninja, Version 2.2.2487 demo
4. Objdump, Version 2.34

This selection is an arbitrary selection of state-of-the-art disassembly and decompilers. All tools have different disassembly and decompilation processes, intermediate Languages, and different decompilation algorithms. Ghidra Rohleder [2019], and Binary Ninja are relatively new and not that long available to the public. Ghidra is open-source and made available by the National Security Agency (NSA)[3]. Binary Ninja[4] was developed by hacker group Vector 35 and is based on their decompilation tooling used for CTFs tournaments. JEB[5] is created and maintained by Pnfsoftware and supports a variety of processor architectures. Pnfsoftware was founded in 2013. Ghidra is open-source in contrast to Binary Ninja and JEB, both commercial applications. However, Binary Ninja and JEB both have a freeware/community version available with limitations. The last disassembly tool in this list is Objdump [6]. Objdump is part of the open-source GNU Binutils collection. This collection contains a collection of binary tools. Objdump uses the Binary File Descriptor library for low-level file information and the opcodes library for disassembly. All four tools have in common that they support the decompilation of x86-64 Windows PE and the Unix/Linux ELF binaries. These tools are well known and frequently used in the industry. With this selection, we have a diverse group of suitable and state-of-the-art disassembly and decompilation tools.

The results of our `switch` statement disassembly survey is noted in Table 3.3 and Table 3.4. These tables show the disassembly results of each decompiler against each combination of the diagnostic test, compiler, and compiler optimisation setting. The superscripts are a cross-reference with the text explaining the verification issue. The disassembly result is, as mentioned before, verified according to our verification definition given in Section 3.2. So, to gather these results, we have examined and verified 256 diagnostic test decompilation results.

One of the first things that can be noticed in the table is that objdump is the only tool that produces only unsound results for all MSVC compiled binaries[1]. Thus objdump creates bogus instructions in his `switch` statement disassembly results. The used disassem-

[3]https://ghidra-sre.org/
[4]https://binary.ninja/
[5]https://www.pnfsoftware.com/jeb/
[6]https://www.gnu.org/software/binutils/

Diag. Test	Compiler	Optimisation	Ghidra	Binary Ninja	JEB	Objdump
C2	GCC	`-Od`	IC[2]	S/C	IC[2]	S/C
		`-Os`	IC	S/C	IC	S/C
		`-O2`	IC	S/C	IC	S/C
	MSVC	`-Od`	S/C	S/C	S/C	US[1]
		`-O2`	S/C	IC[3]	S/C	US
	Clang	`-Od`	S/C	S/C	S/C	S/C
		`-Os`	S/C	S/C	IC[4]	S/C
		`-O2`	S/C	S/C	IC	S/C
C3	GCC	`-Od`	IC[2]	S/C	IC[2]	S/C
		`-Os`	IC	S/C	IC	S/C
		`-O2`	IC	S/C	IC	S/C
	MSVC	`-Od`	S/C	S/C	S/C	US[1]
		`-O2`	S/C	IC[3]	S/C	US
	Clang	`-Od`	S/C	S/C	S/C	S/C
		`-Os`	S/C	S/C	IC[4]	S/C
		`-O2`	S/C	S/C	IC	S/C
C4	GCC	`-Od`	IC[2]	S/C	IC[2]	S/C
		`-Os`	IC	S/C	IC	S/C
		`-O2`	IC	S/C	IC	S/C
	MSVC	`-Od`	S/C	S/C	S/C	US[1]
		`-O2`	S/C	IC[3]	S/C	US
	Clang	`-Od`	S/C	S/C	S/C	S/C
		`-Os`	S/C	S/C	IC[4]	S/C
		`-O2`	S/C	S/C	IC	S/C
R2	GCC	`-Od`	IC[2]	S/C	IC[2]	S/C
		`-Os`	IC	S/C	IC	S/C
		`-O2`	IC	S/C	IC	S/C
	Clang	`-Os`	S/C	S/C	IC[4]	S/C
		`-O2`	S/C	S/C	IC	S/C
R3	GCC	`-Od`	IC[2]	S/C	IC[2]	S/C
		`-Os`	IC	S/C	IC	S/C
		`-O2`	IC	S/C	IC	S/C
	Clang	`-Os`	S/C	S/C	IC[4]	S/C
		`-O2`	S/C	S/C	IC	S/C
R4	GCC	`-Od`	IC[2]	S/C	IC[2]	S/C
		`-Os`	IC	S/C	IC	S/C
		`-O2`	IC	S/C	IC	S/C
	MSVC	`-Od`	S/C	S/C	S/C	US[1]
		`-Os`	S/C	S/C	S/C	US
		`-O2`	S/C	S/C	IC[5]	US
	Clang	`-OS`	S/C	S/C	IC[4]	S/C
		`-O2`	S/C	S/C	IC	S/C

Table 3.3: Diagnostic test disassembly results of the disassembly tools 1/2.

S/C = sound and complete
US = unsound
IC = incomplete

Diag. Test	Compiler	Optimisation	Ghidra	Binary Ninja	JEB	Objdump
S2	GCC	-Od	IC[2]	S/C	IC[2]	S/C
		-Os	IC	S/C	IC	S/C
		-O2	IC	S/C	IC	S/C
	MSVC	-Od	S/C	S/C	S/C	US[1]
		-O2	S/C	S/C	S/C	US
	Clang	-Od	S/C	S/C	S/C	S/C
		-Os	S/C	S/C	IC	S/C
		-O2	S/C	S/C	IC	S/C
S3	GCC	-Od	IC[2]	S/C	IC[2]	S/C
		-Os	IC	S/C	IC	S/C
		-O2	IC	S/C	IC	S/C
	MSVC	-Od	S/C	IC[6]	S/C	US[1]
		-O2	S/C	IC	S/C	US
	Clang	-Od	S/C	S/C	S/C	S/C
		-Os	S/C	S/C	IC[4]	S/C
		-O2	S/C	S/C	IC	S/C
S4	GCC	-Od	IC[2]	S/C	IC[2]	S/C
		-O2	IC	S/C	IC	S/C
	MSVC	-Od	S/C	IC[3]	S/C	US[1]
		-O2	S/C	IC	IC[5]	US
	Clang	-Od	S/C	S/C	S/C	S/C
		-O2	S/C	S/C	IC[4]	S/C

Table 3.4: Diagnostic test disassembly results of the disassembly tools 2/2.

S/C = sound and complete
US = unsound
IC = incomplete

bly algorithm causes this. For objdump, this is a linear disassembly algorithm. The main characteristic of this algorithm is that it will parse all bytes consecutively, and the problem herein is that not all bytes are instructions. MSVC's jump tables, which are placed in the text section, are interpreted as instructions, not as data. Therefore this results in unsound instructions. All other disassembly tools used in this section are recursive. Recursive disassembly/decompilation algorithms do take the control flow into account. So when an x86-64 machine instruction or data is created, it is because there exists a static path that leads to that particular piece of code or data. So a tool with a linear disassembly algorithm will more often produce unsound results than a tool with a recursive algorithm.

Another thing that catches the eye is that both *Ghidra* and *JEB* are not able to correctly disassemble any diagnostic test compiled with GCC[2]. All GCC results for those two mentioned disassemblers were incomplete. Despite the fact of GCC's frequent use in real-life. However, there is a distinction between the produced results and the point from which the disassembly fails. *Ghidra* finds it hard to determine GCC jump-table implementation correctly for each diagnostic test. So the disassembly stops at the `JMP` instruction of the `switch` statement. So all code blocks that relate to the `switch` statement are left out, i.e., the result is incomplete. The same counts for the tests that GCC compiles with no optimisation setting `-Od`, *JEB* also find it hard to determine the jump-table. Of the diagnostic tests that are compiled with the optimisation settings `-Os` and `-O2`, JEB has not been able to disassemble the main function at all. JEB interpreted the main function as data and did not recover the jump-table in the `rdata` section.

Ghidra did produce sound and complete `switch` statement disassembly results for all of the other diagnostic tests compiled with MSVC and Clang. Ghidra could correctly identify the lookup tables and disassemble the case code blocks.

Binary Ninja did disassemble most binaries except some compiled with MSVC[3]. The binaries resulting in incomplete disassembly were the diagnostic tests C2, C3, C4, S3, and S4. For the diagnostic tests C2 and C3 compiled with optimisation setting `-O2`, Binary Ninja has not been able to determine the jump-table and did not disassemble the case code blocks coming after the `JMP` instruction. See Listing 3.3, after the JMP instruction at address `1400010a9` only byte values are given, until a code block that is reachable without the jump table. The jump table is located at address `140001164` till address `14000118C`

For the diagnostic test S3, where MSVC created an index-mapping implementation for the `switch` statement, the problems were related to the secondary index table[6]. For compilation, without optimisation setting `-Od` the secondary table was not of the correct length. For optimisation setting `-O2`, the secondary index table was skipped. Moreover, these missing indexes or tables led to incomplete results.

The disassembly tool *JEB* was unable to disassemble most of the diagnostic tests used for our `switch` statement disassembly verification. As mentioned before, it could not correctly disassemble any of the GCC diagnostic tests. This behaviour also applied to most of the diagnostic tests compiled with Clang[4]. JEB was only able to correctly disassemble the tests compiled with no optimisation setting `-Od`. For the binaries compiled with optimisation other than disabled, JEB could not decompile the main function. This issue with Clang binaries also applies to the binaries compiled with GCC with the same compiler optimisation settings. What also is remarkable about the other disassemblers is that JEB could not deal with `-O2` optimised MSVC binaries with a hybrid `switch` statement implementation[5]. For this particular diagnostic tests, R4 and S4, it could only resolve the

```
140001098  488d1561efffff     lea     rdx, [rel __dos_header]
14000109f  8b8c9a64110000     mov     ecx, dword [rdx+rbx*4+0x1164]
1400010a6  4803ca             add     rcx, rdx
1400010a9  ffe1               jmp     rcx

1400010ab   48 8d 15 c6 11 00 00 33 c9 eb 7c 48 8d 15 c3 11 00 00 b9 01 00
1400010c0   00 00 eb 6e 48 8d 15 bd 11 00 00 b9 02 00 00 00 eb 60 48 8d 15
            b7 11 00 00 b9 03 00 00 00 eb 52
...
140001120   08 00 00 00 eb 0c 48 8d 15 93 11 00 00 b9 09 00 00 00 44 8b c1
            48 8d 0d a4 11 00 00 e8 cf fe ff
140001140   ff 33 c0 48 83 c4 20 5b c3

140001149  488d0d78110000     lea     rcx, [rel data_1400022c8]
140001150  e8bbfeffff         call    sub_140001010
140001155  b901000000         mov     ecx, 0x1
14000115a  ff15980f0000       call    qword [rel exit@IAT]
{ Does not return }

140001160   cc 0f 1f 00 ab 10 00 00 b6 10 00 00 c4 10 00 00 d2 10 00 00 e0
            10 00 00 ee 10 00 00 fc 10 00 00
140001180   0a 11 00 00 18 11 00 00 26 11 00 00 cc cc cc cc cc cc cc cc cc
            cc 66 66 0f 1f 84 00 00 00 00 00
```

Listing 3.3: Binary Ninja misses the lookup table (start address:`140001164`) and thus the switch statement code blocks (address: `1400010ab-140001148`).

first `switch` statement, the subsequent `switch` statements are skipped.

Based on the results in Table 3.3, it is clear that each disassembler has it is strengths and weaknesses for this comparison. For example, Binary Ninja is better at decompiling GCC and Clang binaries. Ghidra is more proficient at the disassembly of Clang and MSVC compiled binaries, and JEB's strength is more in the disassembly of MSVC binaries. Objdump is less suitable for disassembling MSVC binaries due to the mixing of data and instructions in the next section because of its used disassembly algorithm.

The analysis in this section only focuses on disassembly. However, during the analysis also some decompilation errors are found. Notable errors are the use of inline strings, the processing of a `exit` statement and the incorrect decompilation of some of the `switch` statements were not correct decompiled.

An inline string is used when the string literal is directly applied as an operand of assembly instruction. This instead of a pointer to the string in the data section. The compilers frequently use the inline string when optimisation is applied during compilation, with the required condition that a string is short enough to be fitted into the operand. For a compiler to make use of an inline string, the string must be short enough to be fitted in an operand. This implementation makes it a lot harder for a disassembler to correctly identify the correct data type. As a consequence the decompiler cannot identify function signatures correctly for string related functions.

We found that some decompilers find non returning statements hard to analyse during

our analysis. The exit statement is an example of a non-returning function. The exit statement is applied in the diagnostic tests C3, R3, and S3. Both Ghidra and Binary Ninja have frequently skipped the exit statement in their decompilation results.

The decompilation flaws we have encountered during our analysis are missing break statements for each case, missing the `switch` statement at all (while it is correctly disassembled), and wrong case label values.

3.5. SWITCH STATEMENT REAL-LIFE DISASSEMBLY ISSUES

For this section, we will analyse if used disassemblers also encounter `switch` statement disassembly errors while disassembling real-life binaries. The tools used for this analysis are the same disassembly tools as used for the previous test. The decompilation results of the binaries will be checked on the prevalence of lookup tables and the related instructions analysed. This section will discuss our findings on some of the most distinctive disassembly errors.

The real-life binaries we will use are MS Windows DLL binaries. We focus on the DLL files supplied as standard with Windows and compiled with Visual Studios MSVC compiler. We prefer these binaries because they are common, and MSVC's preference for mixing of data and instructions and the use of index mapping as an implementation method for `switch` statements. The three MS Windows DLL executable binaries used for this analysis are:

- ucrtbase.dll: The universal c/c++ run-time library
- kernel32.dll: Win32 base APIs, such as memory management, thread creation
- crypt32.dll: Crypto API, Certificate and Cryptographic Messaging functions

As mentioned before, not all disassembly tools can find jump tables at all. For example, Objdump. Objdump is a simple linear disassembler that interprets everything in the text section as an instruction. No distinction is made between data or instructions. BAP is a more sophisticated disassembler that uses the control flow of an application to separate data and instructions. So in the disassembly result, the table data is not translated into bogus instructions as Objdump does. The downside is that no instructions are available for the jump table's address. So the instructions were the lookup table references to are not in the decompiled result.

The disassembly tools we will be using for this analysis are Ghidra, JEB, and Binary Ninja. All these tools can disassemble jump table based switch statements and MSVC compiled binaries, as we have shown in Section 3.4. We have found three different types of issues during the analysis of the decompilation results that we will be covering in this section. For each of the found issues, we will give one example. The found decompilation issues we have enumerated in Table 3.5. The column isolatable indicates if a diagnostic test is available that gives the same disassembly issue, i.e. we can isolate the disassembly error with a small example.

3.5.1. SECONDARY JUMP TABLES

During our switch statement analysis based on lookup tables of kernel32, we found that Binary Ninja produces different results for the secondary lookup tables as Ghidra and JEB. The heuristic that Binary Ninja uses for analysing secondary lookup tables is deficient. This

	Soundness / Completeness	Disassembler	Isolatable	Real-life examples
Too few entries secondary lookup table	IC	Binary Ninja	Yes	kernel32.dll (0x6b86ba19)
Entries treated as code	US/IC	Binary Ninja	Yes	kernel32.dll (0x6b811a25)
Too complex branching	IC	Ghidra	No	ucrtbase.dll

Table 3.5: Found issues during our real-life switch statement disassembly analysis.

US = unsound
IC = incomplete

```
6b86ba19  uint8_t lookup_table_6b86ba19[0x9] =
6b86ba19  {
6b86ba19      [0x0] =  0x00
6b86ba1a      [0x1] =  0x01
6b86ba1b      [0x2] =  0x02
6b86ba1c      [0x3] =  0x03
6b86ba1d      [0x4] =  0x03
6b86ba1e      [0x5] =  0x03
6b86ba1f      [0x6] =  0x03
6b86ba20      [0x7] =  0x03
6b86ba21      [0x8] =  0x03
6b86ba22  }

6b86ba22        03 03 03 03 03 03    ......
6b86ba28  03 03 04 01 01 05 05 01  ........
6b86ba30  06 03 06 02 07 08 01 01  ........
6b86ba38  01 00                    ..
```

Listing 3.4: Binary ninja disassembled secondary jump table. The secondary lookup table is too short

deficiency results in secondary jump tables not being of the correct length. This result can be isolated with the diagnostic test S4 compiled with MSVC and optimisation setting `-Od` or `-O2`. This imperfection can cause completeness errors. This impacts completeness because binary values are missing in the result.

This deficiency is presented in Listing 3.4 created with Binary Ninja. The byte values placed right after the lookup table (addresses 0x6b86ba22 up to 0x6b86ba40) are not included in this table. They should also be part of the lookup table. The lookup table shown in the listing is the secondary lookup table or index table of a `switch` statement implemented as an index mapping. The size of this index lookup table should be 33 instead of 9.

3.5.2. LOOKUP TABLE ENTRIES TREATED AS CODE

This error is partly related to the previously presented error, secondary jump tables 3.5.1. This is because the disassembled jump table is also of the incorrect length. Only this time, the jump table entries are disassembled as instructions. The erroneous behaviour is related

```
6b811a20  uint8_t lookup_table_6b811a20[0x5] =
6b811a20  {
6b811a20      [0x0] =  0x00
6b811a21      [0x1] =  0x01
6b811a22      [0x2] =  0x04
6b811a23      [0x3] =  0x04
6b811a24      [0x4] =  0x04
6b811a25  }

int32_t __convention("regparm") sub_6b811a25(int32_t arg1) __noreturn
6b811a25  arg1.b = arg1.b + 2
6b811a27  arg1.b = arg1.b + 3
6b811a29  *nullptr
6b811a2b  *nullptr
6b811a2d  breakpoint

6b811a2e  cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
```

Listing 3.5: Binary Ninja disassembled jump tables. Data is interpreted as instruction.

to the index mapping `switch` statement implementation. So the issue can be isolated with diagnostic test S4 compiled with MSVC. This behaviour affects both *soundness* and *completeness* of the result. Soundness because there are instructions in the result that are not present in the binary, and completeness because the jump table entries are lost.

In Listing 3.5 a disassembly result can be seen where the data that should be part of the lookup table is interpreted as code. The address range of the original table is `6b811a20` to `6b811a2e` so the size of the lookup table should be 14. In this disassembly result the size of the table is 5 and the rest of the table indices is interpreted as bogus code, start address `6b811a25` to `6b811a2e`. Binary Ninja even creates an extra function `sub_6b811a25` at address `6b811a25`.

3.5.3. BRANCHING TOO COMPLEX

Disassembly of `switch` statements gets, in some cases, too complex for Ghidra's disassembly analysis. In such cases, Ghidra comes with a warning in the decompilation result. This warning is as follows:
`/* WARNING: Could not recover jump table at 0x1006a383. Too many branches */`
This error results the indirect jump being treated as a function call instead of a `switch` statement. Because Ghidra also uses a recursive control flow analysis, it follows the application control flow, resulting in undecompiled bytes and thus instructions. This is because the possible control flow of the switch statement is not followed. Also, the `switch` statement itself is being omitted by Ghidra. This results in the incompleteness of the produced result.

As an example see Listing 3.6 and Listing 3.7 which are extracted from Ghidra's disassembly result of the binary ucrtbase.dll. Like the previous real-life errors, it is again an index mapping switch statement implementation that causes a disassembly error. In Listing 3.6 can be seen that both lookup tables, the jump table and the index table, are not identified as such by the disassembler. The jump table starts at address `1006ae9f` and ends at

```
        PTR_DAT_1006ae9f    XREF[1]:FUN_10069bdb:1006a383(R)
1006ae9f c4 a3 06 10                addr        DAT_1006a3c4 = 89h
1006aea3 be                         ??          BEh
1006aea4 a3                         ??          A3h
1006aea5 06                         ??          06h
1006aea6 10                         ??          10h
        DAT_1006aea7    XREF[1]: FUN_10069bdb:1006a37c(R)
1006aea7 00                         ??          00h
1006aea8 01                         ??          01h
1006aea9 01                         ??          01h
1006aeaa 00                         ??          00h
1006aeab 01                         ??          01h
1006aeac 00                         ??          00h
1006aead 00                         ??          00h
1006aeae 01                         ??          01h
```

Listing 3.6: Ghidra analysis fails to detect lookup tables.

```
                    DAT_1006a3be                                    XREF
                        [1]:     1006aeaf(*)
1006a3be 83                         ??          83h
1006a3bf 4d                         ??          4Dh    M
1006a3c0 f0                         ??          F0h
1006a3c1 ff                         ??          FFh
1006a3c2 eb                         ??          EBh
1006a3c3 03                         ??          03h
                    DAT_1006a3c4                                    XREF
                        [1]:     1006ae9f(*)
1006a3c4 89                         ??          89h
1006a3c5 75                         ??          75h    u
1006a3c6 f0                         ??          F0h
```

Listing 3.7: Ghidra fails to disassemble into instructions due to incorrect jump-table.

address `1006aea3`. The start address of the index table can be found directly below the jump table at start address `1006aea7`. Thus the jump table contains two addresses which both reference a different basic block. The two addresses that the jump table contains, or should contain, are address `1006a3c4` and address `1006a3be`. Both addresses and the basic blocks can be found in Listing 3.7. As can be seen in this listing is that Ghidra labels both addresses as data instead of instructions and does not disassemble the bytes.

The failure of detecting the lookup tables and thus producing a `switch` statement will result in basic block being referenced to (address `0x1006a3be` and `0x1006a3c4`) is not correctly detected as instructions and being treated as data. See Listing 3.7.

4

POINTER RECOGNITION ISSUES

A pointer holds a pointer value that is an address to another piece of memory. The memory where it references can be instructions or data. So pointers are memory locations ('variables') that hold the address value of a code block (e.g. function), data or another pointer(pointer to a pointer). A pointer can also be used in arithmetic operations like incrementing or decrementing. These operations are used, for example, to manipulate the index of an array. Because a pointer address is a value, the disassembler handles it as data.

In an x86-64 binary, the location of the pointer variable, i.e. memory location of the pointer value, can be included in the text section or one of the data sections. The pointer value can be defined as a variable or as a constant. When a pointer value is declared as a constant, it will be included in the text or one of the read-only data sections. As mentioned before, it is allowed in the x86-64 architecture to mix data and instructions in the executable section. If the pointer value is a variable, the pointer value can be included in a writable data section only. Such as the writable data section (global variables), the stack or the heap. So a pointer value can be included in almost all data or text related segments of the application, depending on the needed characteristics.

As mentioned earlier, a pointer holds a value that addresses another piece of memory. It can be located in almost all data sections or the binary's text section. This pointer value is actually often an immediate address value. These pointers can also occur as a jump table, pointer to pointer, or a data structure containing a pointer. One of the challenges a disassembler faces is that it has to determine if raw, immediate constants actually constitute pointers. If it wrongly decides that a certain constant is a pointer, this may lead to unsoundness because the disassemblers result can include an unreachable address. If the disassembler wrongly decides that a certain constant is not a pointer, this may lead to incompleteness because the disassembler can miss a reachable address in the result. Thus for a disassembler to correctly disassemble all instructions, it also has to analyse the related data bytes correctly to determine if it contains a possible address.

This section will first introduce the definition of soundness and completeness using the pointer recognition analysis. Second, we create a diagnostic test and analyse one particular case of a failed disassembly. Third, we will discuss one real-life issue found during the disassembly analysis of crypt32.dll.

4.1. Soundness and completeness of pointer recognition

The heuristics a disassembler uses to determine how to interpret data for the indirect control flow are disassembler depended. The result of this determination and thus the used heuristics influence both the soundness and completeness of the produced result. The analysis is complete when an immediate value is a pointer, it is recognised as such, and disassembly considers the address pointed to as reachable. The analysis is sound if an immediate value is recognised as a pointer, then it is a pointer. The analysis is unsound if an immediate value is recognised as a pointer wrongly, and thus the disassembly considers an unreachable address as reachable. so the raw data analysis of possible pointers used in indirect control flow analysis affects both the result's soundness and completeness.

We will try to explain soundness and completeness making use of Listing 4.1. The example consists of two functions, a function `main` and a function `foo` placed in the `text` section of the binary. The main function contains a `jump` instruction at address `0x4050`. The address where is jumped to is assigned to register `ax` prior to the `jmp` instruction. The `data` section contains a pointer value to the function foo at address `0x9000` and a value at the address `0x9002` that is not a pointer value but data.

Assume that register `ax` used in the indirect `jmp` instruction is calculated before the jump. The only valid value of `ax` is the address `0x5000` using the pointer at address`0x9000`. The address value `0x9002` contains not a pointer value

The disassembly result is *complete* if it contains the function foo. The result is *incomplete* if it is missing the function foo. So the result is complete if *all* reachable functions are in the result.

The disassembly result is *sound* if it only contains the function foo. The disassembly result is *unsound* if it contains a function at address `0x6000`. So the result is sound if *only* reachable instructions are in the result.

4.2. Pointer recognition

In this section, we will create a small diagnostic test to analyse the recognition of pointer structures. First, we will create and explain a diagnostic test. Second, we will perform an analysis of an incorrect disassembly result.

4.2.1. Diagnostic test

To analyse pointer structure recognition, we will first craft a small diagnostic test. A pseudocode example of this diagnostic test is placed in Listing 4.2. This piece of source code contains an array of pointers named `pointer_values`. Each index of the array contains a `struct` with two pointers, a pointer to a string and one to a function. The pointer values are assigned consecutively. For each index, first, a string pointer to pointer is created. Second, a function is assigned. A schematic overview of the pointers is given in Figure 4.1. The `pointer_values` array is iterated over in the `main` function and for each index the contained string value is compared to the input string. When an input string matches, the function related to this input is executed by calling the function pointer in the array.

```
.text
        main
    0x4000  push rbp
        ...
    0x4050 jmp  word ptr [ax]
        ...
    0x4100  return

        foo
    0x5000 push rbp
        ...
    0x5100 return

    0x6000 ..

.data

    0x9000 0x5000 ; Is a pointer value.
    0x9002 0x6000 ; Is data, not a pointer value.
```

Listing 4.1: Example of a pointer value in a data section

4.2.2. DIAGNOSTIC TEST DISASSEMBLY

To perform the analysis, we have compiled the diagnostic test with MSVC and optimisation setting `-O2`. In this diagnostic test, we have disassembled with the same disassemblers as used for the `switch` statement disassembly analysis. So we have analysed the results of the following disassemblers, Ghidra, JEB, and Binary Ninja.

We will discuss one of the failed diagnostic test disassembly results. The disassembler that produced the issue is the disassembly tool JEB. When JEB disassembles the diagnostic test, it produces an incomplete disassembly result. So the result lacks *completeness* concerning the input binary. Listing 4.3 shows us the missing detected pointer values of the array `pointer_values`. Only the first pointer value is found at address `400022E8`, the pointer values hereafter are displayed as raw byte values. As a result of this interpretation error only `function_A` is found at address `140001070` in Listing 4.4. The other functions, following right after `function_A`, are not identified. The start addresses of the skipped functions are for `function_B` at `140001090` and for function_C at `1400010B0`. As a result, the raw byte values are shown in the listing instead of instructions for these particular functions.

4.3. POINTER REAL-LIFE RECOGNITION ISSUES

In this section, we discuss in detail one example that explains the error found in the decompilation result. When pointers are not recognised correctly by the decompiler, as mentioned before, mainly two issues can arise. First, the pointer value itself and the function it references are left out in the result. Second, a value is incorrectly interpreted as a pointer, and a non-existing function is added to the produced disassembly result. Beyond this, in some cases, when the pointer address is placed in the text section of the binary, the pointer value itself can be interpreted as one or multiple instructions. This interpretation error also leads to soundness and completeness issues. Completeness because the basic block where

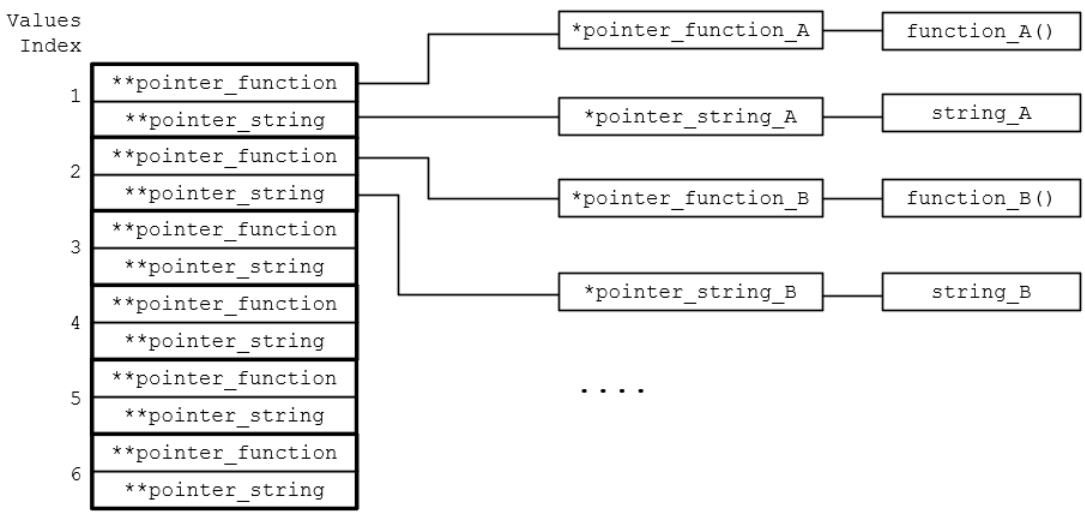

Figure 4.1: schematic view of the pointer to pointer array

the pointer is referencing is left out in the result, and soundness because not existing instructions are added to the disassembly result. As a result, this can lead to the desynchronisation of the disassembler. The desynchronisation happens when a byte of an existing instruction is taken by the preceding (not existing) instruction, which can lead to another wrongly disassembled instruction because of the wrong start address being used, and so on. As a result, will desynchronisation lead to soundness issues. So a not correctly identified pointer address can cause havoc on the disassembly results.

An example of an incorrectly recognised pointer is found in Ghidra's disassembly result of the crypt32 DLL. Ghidra indicates pointers with by the assembly `addr` operator in the disassembly result. This can be seen in the first two rows of the following **crypt32.dll** Listing 4.5, Address `5cf016f8`, and `5cf016fc`. Several pointers coming directly hereafter, at start address `5cf01700` up to address `5cf01700`, are incorrectly interpreted as code instead of data e.g pointer values. So the instructions shown in this disassembly result are bogus instructions.

In the found example, the pointers are placed in the text section of the binary. The decompiler interprets them as code instead of pointer values. When we take a look at the relocation section, we can verify that they are indeed pointer addresses, Listing 4.6 and not code like in Ghidra's result. The relocation table contains all addresses that will be updated when the binary needs to be loaded at a different relative virtual address (RVA) than compiled. The data structure contains pointers to pointers that point to strings and functions. These strings and functions are related to cryptographic processing. The pointer structure we show is probably part of a data structure that contains a list of strings with a corresponding function for processing.

```
string_A = "A"
string_B = "B"
...

pointer_string_A = &string_A
pointer_string_B = &string_B
...

pointer_values[] = {
        {
                .pointer_function = function_A,
                .pointer_string = &pointer_string_A
        },
        {
                .pointer_function = function_A,
                .pointer_string = &pointer_string_A
        },
...
}

void function_A() {
        print("Function A ")
}
void function_B() {
        print("Function B ")
}
...

void main(input) {

        for (int index = 0; index < values.size; index++) {
                if (equal(values[index].pointer_string,input)) {
                        values[index].pointer_function();
                }
        }
        return
}
```

Listing 4.2: Pseudo code example of the pointer diagnostic test

```
.rdata:1400022E8    gvar_1400022E8   dq 140004070h
                                ; xref: sub_140001130+40h (data-adv)
.rdata:1400022F0    db 90h, 10h, 0, '@', 1, 0, 0, 0, "`@", 0, '@', 1, 0,
   0, 0
.rdata:140002300    db B0h, 10h, 0, '@', 1, 0, 0, 0, "8@", 0, '@', 1, 0,
   0, 0
.rdata:140002310    db D0h, 10h, 0, '@', 1, 0, 0, 0, "X@", 0, '@', 1, 0,
   0, 0
.rdata:140002320    db F0h, 10h, 0, '@', 1, 0, 0, 0, "H@", 0, '@', 1, 0,
   0, 0
.rdata:140002330    db 10h, 11h, 0, '@', 1, 0, 0, 0, "x@", 0, '@', 1, 0,
   0, 0
```

Listing 4.3: JEB disassembly result of disassembled diagnostic test pointers. The pointer_values array is not disassembled correctly.

```
====================================
; ROUTINE: function_A

.text:140001070  function_A        proc
.text:140001070
.text:140001070      mov    r9, qword ptr ds:[r8]
.text:140001073      mov    r8d, edx
.text:140001076      mov    edx, ecx
.text:140001078      lea    rcx, qword ptr ds:[aFunction_A] ;a pointer to
   the Function_A string
.text:14000107F      jmp    printf
.text:14000107F
.text:14000107F  functionA         endp
-------------------------------------

.text:140001084     db CCh, CCh, CCh, CCh, CCh, CCh, ...
.text:140001090     db 'M', 8Bh, 8, 'D', 8Bh, C2h, 8Bh, ...
.text:1400010A0     db 'l', FFh, FFh, FFh, CCh, CCh, CCh, ...
.text:1400010B0     db 'M', 8Bh, 8, 'D', 8Bh, C2h, 8Bh, ...
.text:1400010C0     db 'L', FFh, FFh, FFh, CCh, CCh, CCh, ...
...
```

Listing 4.4: JEB disassembled pointers of test code. Some functions are missing

```
; Correcly idenitfied pointer values:
5cf016f8 ec cd f0 5c              addr       DAT_5cf0cdec = 31h
5cf016fc 30 82 f8 5c              addr       FUN_5cf88230

;Incorrecly identified pointer values:
5cf01700 d4 cd                    AAM        0xcd
5cf01702 f0                       LOCK
5cf01703 5c                       POP        ESP
5cf01704 20 85 f8 5c c8           AND        byte ptr [EBP + 0
   xcdc85cf8]=>DAT_cdceb9f0,AL
         cd
....

5cf0171b 5c                       POP        ESP
5cf0171c 00 f2                    ADD        DL,DH
5cf0171e f8                       CLC
5cf0171f 5c                       POP        ESP
    LAB_5cf01720 XREF[1]:     FUN_5cf43490:5cf434eb(*)
5cf01720 01 00                    ADD        dword ptr [EAX],EAX
5cf01722 00 00                    ADD        byte ptr [EAX],AL
5cf01724 e0 0f                    LOOPNZ     LAB_5cf01735
5cf01726 f4                       HLT
5cf01727 5c                       ??         5Ch    \
    LAB_5cf01728 XREF[1]:     5cf0174c(j)
5cf01728 02 00                    ADD        AL,byte ptr [EAX]
5cf0172a 00 00                    ADD        byte ptr [EAX],AL
5cf0172c 10 a1 f8 5c 03           ADC        byte ptr [ECX + 0
   x35cf8],AH
         00
```

Listing 4.5: Ghidra disassembled pointers. Some pointers are interpret as instructions

```
5cf01700        0x3             d4 cd f0 5c
5cf01704        0x3             20 85 f8 5c
5cf01708        0x3             c8 cd f0 5c
5cf0170c        0x3             a0 ef f8 5c
5cf01710        0x3             bc cd f0 5c
5cf01714        0x3             50 f0 f8 5c
5cf01718        0x3             ac cd f0 5c
5cf0171c        0x3             00 f2 f8 5c
5cf01724        0x3             e0 0f f4 5c
5cf0172c        0x3             10 a1 f8 5c
```

Listing 4.6: Binary ninja disassembled jump tables. Data is interpret as instructions

5

Related Work

Paper	Language	Method	Manual Analysis
An Analysis on Java Programming Language Decompiler Capabilities	Java	Testcase source code comparison	✓
An Evaluation of Current Java Bytecode Decompilers	Java	Source code comparison based on software and quality metics	
How Far We Have Come: Testing Decompilation Correctness of C Decompilers	C	Equivalence Modulo Inputs (EMI) Testing	
The Strenghs and Behavioural Quirks of Java Bytcode decompilers	Java	Equivalence Modulo Inputs (EMI) Testing	
This study	C	Diagnostic Test source code comparison	✓

Table 5.1: Enummeration of studies that compare decompilers.

Several recent studies were conducted that target comparison of decompilers that we could also use to compare disassemblers. Table 5.1 enumerates four of those studies. In the first column, the title of the papers is displayed. In the second column, the targeted programming language of the decompiler is shown. The used methods for analysing and comparing the decompiler outputs are stated in the third column. The last column indicates if the used analysis method is based on a manual comparison of the produced result with the input source code. The study is checked with a '✓' if a manual comparison method is used.

As stated in Table 5.1, most studies target the comparison of Java decompilers. This is most likely because of the popularity of the language among programmers. For almost 15

years, Java is already in the top 2 of the Tiobe index for most used programming languages [1]. Also, the decompilation of Java byte code is less complex than the decompilation of x86-64 binaries. For example, the Java bytecode exists with fewer instructions, around 200 [2] vs around 2000 for the x86-64 assembly language [3]. Also, the bytecode file has a different structure, making the decompilation process more convenient.

Each study compares the decompilers based on different inputs and capability aspects. The study of Gusarvos Gusarovs [2018] tests the decompiler with one test case on complex branching, a function with no return type, and some new Java capabilities that were introduced in the lasted Java release at the time of writing. The complex loop consists of a loop with different entry points randomly invoked by using labels and different exit points using break statements. The study of Hamilton et al. uses different test cases Hamilton and Danicic [2009]. Those test cases vary from a simple program such as the Fibonacci algorithm, typecasting and more complex test cases such as optimised bytecode and a small game application. The tested capabilities are typecasting decomposition, an inner class, type inference, try-finally blocks, control flow, exceptions, optimised bytecode, and variable reuse. Liu et al. test the different decompilers using multiple C applications generated with Csmith Liu and Wang [2020]. Csmith is a compiler test tool that randomly generates C applications [4]. The decompilation results are used to find decompilation failures, recompilation failures and decompilation defects. So this study does not target specific decompilation capabilities but tries to find the weak spots of the tested decompilers. The study of Harrand et al. defines three quality indicators for the decompilation process Harrand et al. [2019]. According to the study, those quality indicators are syntactic correctness, syntactic distortion and semantic equivalence. The testing of those quality aspects is based on several real-world open-source software projects Hamilton and Danicic [2009].

All studies use test cases with available source code to compare with the decompilers results. Three of the studies we looked at use special for the study crafted test cases Gusarovs [2018]Hamilton and Danicic [2009] Liu and Wang [2020]. One of these studies uses randomly generated C applications. Only one study uses real-world open-source applications Harrand et al. [2019]. So each study uses a different approach on how the decompilers are tested and mutually compared.

To compare the decompilation tool's capabilities, we have to compare the decompilation results. Half of the studies use the Equivalence Modulo Inputs (EMI) method Le et al. [2014]. EMI is initially developed for the validation of compiler optimisations. This method aims to check if decompiled code behaves the same as the original. This is performed by comparing the inputs and the behaviour of an original program to a recompiled program. This does, however, mean that the result must be recompilable. One study uses one small test case and analyses the result by hand Gusarovs [2018]. Hamilton et al. use software metrics and the semantics and the syntax quality of the produced result as a comparison. The quality scale ranges from 0 to 9 and ranges from a correct decompilation result to fail, no compilation result at all. According to Naeem et al. Naeem et al. [2007] software metrics like code size, conditional complexity, abrupt control flow and the use of local variables are a good representation of the effectiveness of decompilation results.

[1] https://www.tiobe.com/tiobe-index/
[2] https://docs.oracle.com/Javase/specs/jvms/se10/html/jvms-6.html
[3] https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
[4] https://embed.cs.utah.edu/csmith/

The herein mentioned studies show us that the results between the tested decompilers, even for a small test case, can differ a lot. Two of these studies evaluate only semantics and compilability Liu and Wang [2020] Gusarovs [2018] Harrand et al. [2019] and one evaluates semantics, compilability and syntax constructions Hamilton and Danicic [2009]. The results for all studies range from cases that produce correct functioning compilable code to incorrect uncompilable code. However, mostly, they produce uncompilable code which needs human interaction to make it compilable. The studies that use more real-world examples show that no decompiler passes all tests, even for the less complex decompilation of Java bytecode Hamilton and Danicic [2009] Harrand et al. [2019].

Liu et al. take the analysis of the decompilation results even further and tries to find the root causes of the found errors Liu and Wang [2020]. They classify three types of decompilation errors: Decompilation failures (decoding bug), recompilation failures (errors while recompiling), and decompilation defects (semantics is broken in the decompiled result). The analyses of recompilation failures show two key reasons. The first reason is erroneous variable recovery, and the second reason is undefined symbols. Typical decompilation defect issues are found due to: no foolproof type recovery, variable recovery, control flow recovery (caused by wrong type recovery and optimisation bugs), and optimisations to make the decompilation code more readable.

6

Discussion and Conclusion

In this section, we will discuss and conclude our study. To do this, we will first summarise our results. Second, we will discuss the study. We give our study's scope, limitations, assumptions, and weaknesses during this discussion. Third, we will conclude our study. Last, we will talk about possible development ideas in future work.

6.1. Summary of Results

During our `switch` statement analysis, we have performed several steps to perform our analysis process. First, we created diagnostic tests and compiled these with the mentioned compilers. The result of these steps is a binary implementation of the switch statement as included in the diagnostic test source code. Hereafter we analysed and checked the compiler type of `switch` statement assembly implementations. This assembly implementation results from the source code and compiler characteristics together. The last step was to check if several disassemblers were able to disassemble the various lookup table based `switch` statement assembly implementations. At last, we have analysed and verified some real-life issues regarding the disassembly of `switch` statements. In this section, we will summarise our study results.

The characteristics that influence the assembly implementation of a `switch` statement by the compiler are source code and compiler dependent. The essential source code characteristics are the case value set and the number of `case` labels. The compiler characteristics that influence the created assembly implementation are the compiler itself, the used optimisation setting and the switch statement implementation conditions.

We have crafted several diagnostic tests and varied with the earlier characteristics to perform this analysis. For the source code, we have varied the number of `case` labels, the value set and applied some other code constructs in and around the `switch` statement. We used several compilers and varied the optimisation settings to compile the diagnostic tests.

The used compilers choose for the diagnostic test with a low `case` label count, in general, a sequential test implementation. For the diagnostic tests with a consecutive or a sparse value set with a `case` label count greater than three, mostly a lookup table or index mapping variant is chosen. Some compilers choose for the favour size (`-Os`) optimisation

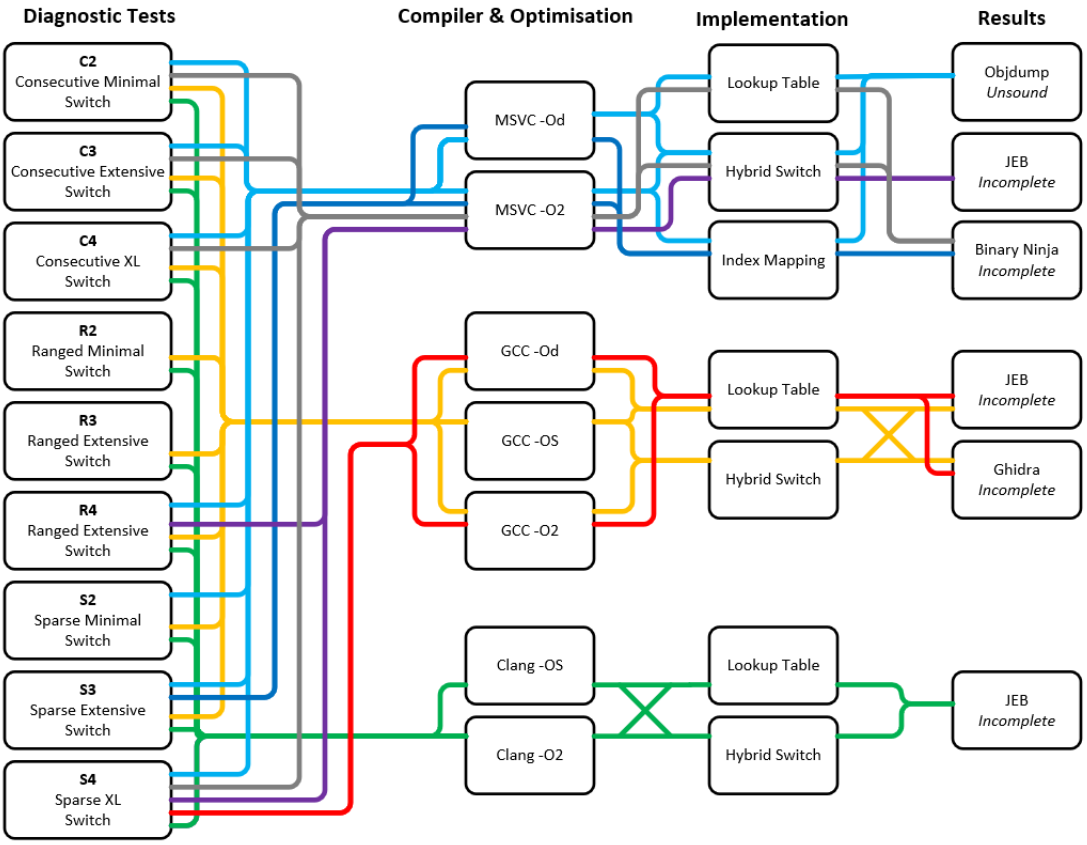

Figure 6.1: Schematic overview of the Switch statement verification process results.

a sequential test because this solution is less memory consuming. For the ranged diagnostic test with a greater switch count, the compilers generally chose to implement a hybrid switch solution. The used sub-implementations for the hybrid variants are mostly based on the lookup table variants.

Among the used compilers is MSVC, the compiler that uses the most variety of implementations between the compiler optimisation settings. It is also the only compiler that creates the index mapping variant. There are only a few differences between the chosen solution between GCC and Clang. For GCC and Clang, the compiler optimisation settings have less influence on the binary implementation than MSVC. GCC and Clang used optimisation settings only to influence the implementation details such as instruction sequence, `if-else` statement sequence or the location of the lookup table.

Figure 6.1 gives an overview of the study results. In this figure, all paths throughout the process (Figure 2.1) are drawn, resulting in an unsound or incomplete disassembly result. The path starts with the created diagnostic test with a different implementation of the defined source code characteristics. The diagnostic test was compiled with the given compiler and optimisation setting. The assembly `switch` statement implementation of the produced executable binary is checked on implementation type. Last, the binary was disassembled, and the disassembly result was verified on correctness. Thus this path consists of the combination of the diagnostic test, the used compiler and optimisation, the resulting `switch` statement and the failed disassembler with result classification.

Not one analysed disassembler has been able to disassemble all `switch` statement diagnostic tests correctly. For each disassembler, there was another type of failure. Thus there was not one particular item of the diagnostic test that was hard to disassemble.

Ghidra had a hard time disassembling all GCC diagnostic test binaries. It could not correctly determine the jump table and produced an incomplete result. As a result the `switch` statement basic blocks were left out.

Binary Ninja fails for some of the MSVC binaries to recognise the jump table at all and produces, in those cases, an incomplete result, omitting the `switch` statement basic blocks. For the MSVC index mapping solution, Binary Ninja used an incorrect heuristic resulting in an incorrect table length. As a consequence, it has left out entries from the table. For MSVC binaries that implemented an index mapping solution and compiled with the optimisation `-O2` the secondary table was left out of the result.

JEB has not been able to disassemble most of our diagnostic tests. Most binaries compiled with GCC and Clang were incomplete. It was caused by not being able to interpret the jump table or not being able to disassemble the whole function containing the `switch` statement at all. For MSVC binaries with a hybrid switch statement implementation, JEB skipped the switch statement following the first switch statement, producing an incomplete result.

Objdump uses, in contrast to the other disassemblers, a static linear disassembly algorithm. Due to the jump table's location in the MSVC binaries, the table is also disassembled into instructions. Consequently, the result of Objdump for all MSVC binaries was unsound.

We have also verified what kind of failures we have encountered in real-life MS Windows DLL executable binaries. Part of this analysis was if we could link a real-life issue to an issue found during the diagnostic test analysis. During the real-life analysis, we have found three

types of issues. The issues that we have found were:

- Too few entries secondary lookup table,
- Lookup table entries treated as code, and
- Too complex branching

The first issue is related to the secondary lookup table of the index mapping implementation. The size of the secondary table is always the same as the lookup table. This heuristic is incorrect because the secondary table only contains indexes of the lookup table, so the table's sizes are not related. In this case, Binary Ninja does not correctly identify the size of the secondary lookup table and, thus, produces an incomplete result with missing entries.

For the second issue, lookup table entries treated as code is related to the previous issue. In this case, the disassembler interprets the skipped lookup table entries as code. Thus Binary Ninja produces an unsound and incomplete result.

The third issue is related to Ghidra. For some cases, Ghidra has not been able to correctly identify the possible code paths related to the `JMP` statement and places a too complex branching warning in the result. This issue is possibly due to an incorrect lookup table determination. The lookup table itself is left out in the results, and the first entry is interpreted as a data pointer. Thus the disassembly result is incomplete.

For the first two issues, we could isolate the cause of the issue with one or more of the diagnostic tests. They were both related to the disassembly result of Binary Ninja. We have not encountered the third issue during our diagnostic test analysis. So we only encountered this issue during the real-life analysis.

During this study, we have also examined some pointer recognition errors. Due to this interpretation error, the disassembly result can contain completeness and soundness issues. Completeness issues arise due to skipped basic blocks or functions because the disassembler misses the pointer address and, therefore, the reference. The soundness issues were found when the address value itself was not recognised as such, and the bytes were disassembled as bogus instructions.

6.2. Discussion

In this section, we will discuss our study. First, we will say something about the scope of our study. Second, we will focus on our study's limitations, assumptions, and weaknesses, and third, we will mention other disassembly correctness issues. Last, we will discuss how we can improve our analysis.

The scope of our study is `switch` statement assembly implementations concerning x86-64 executable binaries compiled in the PE format suitable for MS windows. For the switch statements, in particular, we have limited our study to the reachability of the lookup table based `switch` statement assembly implementations. For real-life examples, we have only looked at MSVC compiled x86-64 DLL executable binaries supplied with MS Windows.

One limitation is that despite a large amount of disassembled binaries, we only have tried a limited number of source code characteristic variations. Also, we have analysed the disassembly results of a relatively small amount of disassemblers and compilers. Another limitation of our study is that we use static disassembly results for our analysis. So it is possible that the used disassembly results do not reflect the run-time behaviour of the binary due to side effects and other not obvious possible code paths.

An assumption we have made is that if a `switch` statement basic block was reachable, the prior switch conditions were also met—preceding conditions such as switch value and corresponding case value and range check.

A weakness of our study is that the disassembly results are manually checked with the corresponding diagnostic test source code. So a manual translation is done between the disassembly results and the higher language input. A manual analysis of assembly code can be complicated and prone to errors due to the complexity of the x86-64 assembly language.

Other issues that can target the correctness of disassembly results are the recognition of pointers, overlapping instructions or data in the `text` section. A pointer can point to data, functions or a basic block. When pointers are not recognised correctly, this can lead to correctness issues in the disassembly result. So a misinterpreted data pointer can result in invalid instructions. When a pointer to a function or basic block is misinterpreted, this can lead to missing instructions. When a binary uses overlapping instructions, this will also result in correctness issues due to missing instructions. Overlapping instructions are used for obfuscation or to jump over instruction prefixes. This also means that the disassembly result will be incomplete. As already shown during our study, data in the `text` section, which MSVC is notorious for, can lead to incorrectness of the result.

The manual approach of the diagnostic test disassembly analysis is very labour intensive and error-prone. To simplify and (partially)automate this manual analysis, some other alternatives mentioned in the related work section can be used. One example is making use of Equivalence Modulo Inputs (EMI) Testing. EMI is developed to validate compiler optimisations and check if the optimised code behaves the same as the original. In our case, it can check if the disassembled code behaves the same as the source binary.

6.3. Conclusion

During our diagnostic test disassembly analysis, we found that each disassembler has its strengths and weaknesses regarding `switch` statement disassembly. So they produce varying disassembly results for each of the used compilers. The results depend on the used disassembly algorithm and the used heuristics. We have found both incomplete and unsound results. Because most of our disassemblers use a static recursive algorithm, the most found issue is related to incompleteness or the result, thus missing assembly instructions in the produced result. The most difficult for a disassembler is correctly identifying and interpreting the lookup table. The disassembly errors we found regarding a `switch` statement lookup table implementation are the disassembler is unable to identify the lookup table at all, or the disassembler is unable to determine the correct size of the lookup table.

6.4. Future work

Future research could examine the verification of decompilation `switch` statement results. This decompilation process is even more complex than the disassembly steps analysed during this study. Extending this study with the process of lifting the disassembly result into the decompilation will give insight into how well the state-of-the-art decompilation algorithms can handle `switch` statements. Also, the higher level decompilation result can be used for analysis, validation or improvement of executable binaries.

Another item that can be studied for future research is the decompilation of the calling

convention used by MSVC. There are in total four different calling conventions with different passing of parameters and stack cleanup rules. This in contrast to most other compilers. It would be interesting to see if decompilers can correctly handle all MSVC calling conventions.

BIBLIOGRAPHY

Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016. 2, 13

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011. 2, 8

David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 353–368, 2013. 2, 7, 8, 9

Konstantins Gusarovs. An analysis on java programming language decompiler capabilities. *Applied Computer Systems*, 23(2):109–117, 2018. 4, 37, 38

James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 129–136. IEEE, 2009. 4, 37, 38

Nicolas Harrand, César Soto-Valero, Martin Monperrus, and Benoit Baudry. The strengths and behavioral quirks of java bytecode decompilers. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 92–102. IEEE, 2019. 1, 4, 37, 38

Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014. 2, 5, 37

JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011. 8

Zhibo Liu and Shuai Wang. How far we have come: testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487, 2020. 4, 37, 38

Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016. 2

Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999. 8

Nomair A Naeem, Michael Batchelder, and Laurie Hendren. Metrics for measuring the effectiveness of decompilers and obfuscators. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 253–258. IEEE, 2007. 5, 37

Roman Rohleder. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 77–78, 2019. 20

Roger Anthony Sayle. A superoptimizer analysis of multiway branch code generation. In *Proceedings of the GCC Developers Summit*, pages 1–16. Citeseer, 2008. 11

Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *International Conference on Software Engineering and Formal Methods*, pages 247–264. Springer, 2020. 1, 2, 7, 8

Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017. 1

Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 522–536. Springer, 2011. 1

Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. 9