universitätbonn

Challenges of Recovering Binary Disassembly

Felix Bilstein

Seminar Paper

Examiner: Prof. Dr. Peter Martini
Advisor: Daniel Plohmann
Institute for Computer Science IV
Work group Communication Systems
University Bonn

Statement of Authorship

I hereby confirm that the work presented in this seminar paper has been performed and interpreted solely by myself except where explicitly identified to the contrary. I declare that I have used no other sources and aids other than those indicated. This work has not been submitted elsewhere in any other form for the fulfilment of any other degree or qualification.

Bonn, October 8, 2019

Felix Bilstein

Abstract

Disassembly is the foundation of various binary analysis techniques and reverse engineering. Analysis of binary files is an important topic for security researchers from vulnerability researching to binary patching. In this paper we present the current state of research regarding to disassembly derivation from compiled binary files from high level programming languages such as C or C++. We provide an outline about the two primary disassembly techniques linear sweep and recursive traversal approaches as well as a short introduction to executable binary file structure and reverse engineering in general. We discuss complex cases where simple disassembly approaches are limited through inherit boundaries and present modern techniques tackling these limitations.

Contents

1 Introduction

Today's world is based to a great degree on software. While applications can be distributed as source code or in binary form, the latter is common practice in proprietary software deployment. Gaining information about the internals of software in compiled form as binaries is a challenging task which ultimately needs sophisticated tools. A highly valuable as well as fundamental tool for reverse engineering of binaries are disassemblers.

The need for reverse engineering and therefore disassembly recovery is an important area in the scientific community as well as in the industry. There are various use cases for disassemblers in practical applications. Third party audits in the information security field where source code is not necessarily available to auditors are not possible without proper disassembly of the target application. Vulnerabilities found in software where source code is not be available need to be patches by third-parties being unable to recompile the vulnerable code [Sot06].

But also in the software development there is a need for binary patching and sometimes even reverse engineering, for example when the source code is simply lost. Although it is hard to proof that a company lost its source code for a software component it is at least possible to check how the software was altered during updates. Analyzing the application and reverse engineering functions affected by any updates lead to the assumption that patching binaries is practice although it is usually a time consuming and complex task which requires expert knowledge [Sot06]. ACROS Security detected such a binary patch when they analyzed Microsoft's upgrade of an Equation Editor component in 2017 when fixing the vulnerability "CVE-2017-11882" [Kol17].

Reverse engineering of software is a complex and well researched topic in academia [LD03] [ZS13] [ACvdV+16] [ASB17]. Several tools and approaches evolved over the years but even modern disassemblers and frameworks have weaknesses or lack in discovering the correct instructions or code.

In this work we present the current state of research with regard to disassembly generation for reverse engineering of binary files and machine code. After an introduction to the general methodology of compilation and code recovery, in Section 3 we discuss the primary disassembly approaches linear and recursive algorithms and give a short summary about their advantages and challenges combined with a selection of current issues of modern disassemblers developed in academia.

To integrate the current disassembly approaches into modern requirements we give a short outlook about related work and use cases such as binary instrumentation as well as binary lifting and rewriting. Section 4 provides more context for current research topics related to disassembly and the challenges of the process. Disassembly generation from binary files is an essential foundation for following analysis steps.

2 Basic Notions

To provide a certain amount of background information to the reader we present the concept of reverse engineering as a defined process and offer a brief overview of the basics of disassembly and reverse engineering in general.

In the following Section 2.2 we will give a basic introduction to the process of software compilation and reverse engineering itself, the importance of proper disassembly and the challenges of disassembling software in its binary form. In this overview the building process in low level languages is shown in an abstract manner. Besides the structural analysis of binaries like i.e. searching for file types, header, sections and other disassembly is the foundation of binary analysis and reverse engineering. Further methods like decompilation depend on the quality of the recovered disassembly from a given binary.

Section 2.1 presents the following steps after compilation of software and how the resulting machine code is further processed into machine-readable binary files. Binary files consist of a header and a body and are created by the linker. The header contains various important information for the operating system which are necessary to load and execute the binary file while the body consists of code and data sections. The information in the binary header are valuable for disassemblers since they can take advantage of the information to provide disassembly for the appropriate platform.

2.1 Overview: Compilation and Linking

The compilation involves preprocessing, compilation, assembly and linking of the source code files into executable machine code [TB14]. Since we focus on disassembly in this paper the decompilation steps described in Section 2.2 and Figure 3 are out of scope. Therefore it is important to understand the details of the compilation process as well as how the machine code is placed in the executable binaries. Disassemblers can take advantage of the information that are stored in binary files to cover the instructions more precisely. Figure 1 shows the details of compilation and linking schematically.

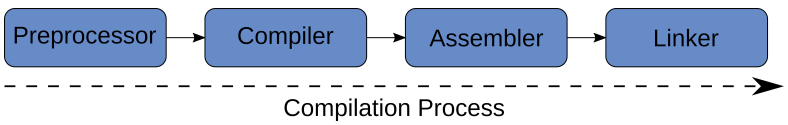

Figure 1: *The process of compilation, adapted from Tanenbaum and Bos [TB14], illustration based on [TB14].*

The source code of programs in high level languages like C is compiled and linked into executable files. The imports in the source code are organized by the preprocessor [TB14]. After that, the program is compiled by a compiler into assembler instructions and labels. The resulting machine code files of the program are linked into a machine readable executable for the target operating system and architecture. Modern compiler suites such as GCC or Clang are able to perform all four steps in one call.

When programs are reverse engineered it is usually assumed that the structure of the analyzed program is an executable file format such as e.g. ELF (Executable and Linking Format) or PE (Portable Executable). These file formats usually consist of a header and a body segment. The header contains several fields such as e.g. the entry point of the program while the body segment contains sections where the structures from the linking process are stored, for example the text and read-only data [LD03]. The information stored in the binary header can be used by the disassembler to detect the target architecture or platform and helps it to produce more precise disassembly. Since the disassembler does not need to assume certain properties of the internal structure of the given binary it can focus on the correct sequences in the file. Depending on the location of code and data or at which location possible jump tables, data reference tables or import are stores, it can generate disassembly only for the appropriate sections.

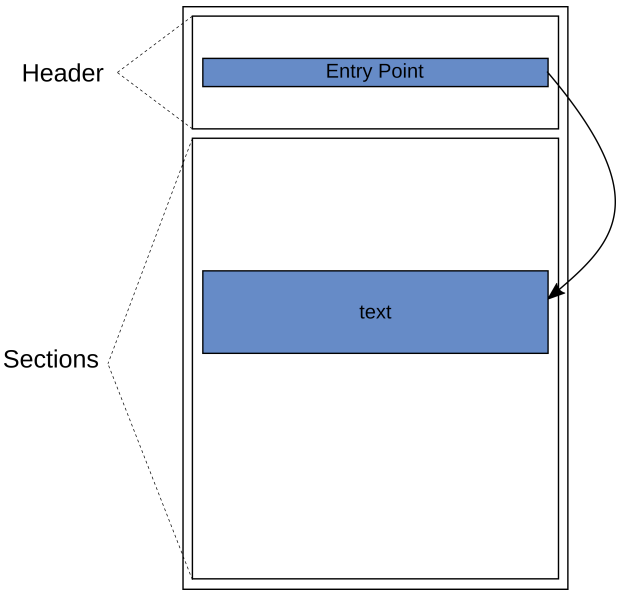

Figure 2: *Binary files, adapted from Linn and Debray [LD03], illustration based on [LD03] and [Lev99].*

Figure 2 illustrates the structure of an executable file. The illustration shows both parts of the binary file; the first segment contains typically the header while the sections of the binary are ordered in the body of the binary. The entry point of a program is the address where the program is initially being executed. The process of disassembly in the context of binary files refers to disassembling the assembly instructions of executable sections in binary files. Typically this is the text section which contains the instructions of a program and is therefore a major target when disassembling binary files. [LD03] [Lev99]

2.2 Contrasting Compilation and Reverse Engineering

During the development of software in programming languages like C or C++, developers create and edit source code files which are compiled into executable binary files. To gain information about the inside of binary files the applications are reverse engineered. Reverse engineering is typically a time-consuming task that requires expert knowledge. The process of compilation and reverse engineering described by *Linn and Debray* [LD03] is illustrated in Figure 3.

Building software from source code files requires several steps; First, the source code is parsed by the compiler and converted into a syntax tree structure. The syntax tree consists of conditions, branch instructions, constants and variables. A syntax tree is constructed for every statement or sequence, although the concrete structure and behavior of the compiler is implementation specific. The syntax

tree is the basis of the control flow graph, which is built by the compiler for further optimization and code generation. The compiler translates the code blocks and control flow graphs into assembly code. Assembly code is the representation of a program for a specific target architecture or platform. The instructions are based on an instruction set architecture (ISA) and highly depend on the target processor. This assembly code can be assembled by certain assemblers and frameworks into machine code. Figure 3 shows the steps of software compilation schematically. [LD03]

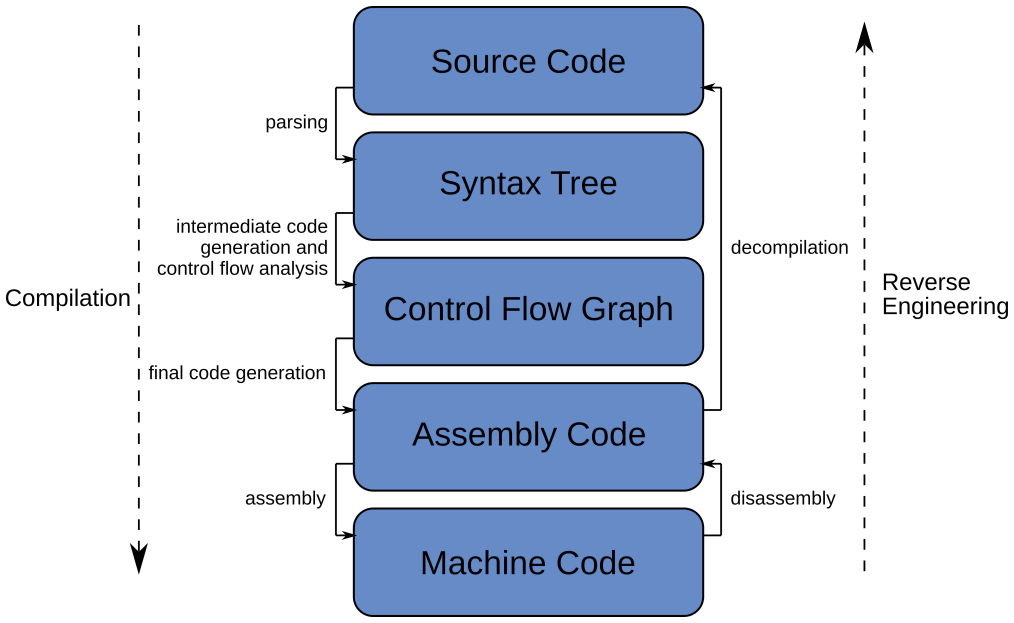

Figure 3: *The process of reverse engineering in detail and as a defined work flow by Linn and Debray [LD03], illustration based on [LD03].*

Details of reverse engineering of binary files is covered by *Linn and Debray* in their paper *Obfuscation of executable code to improve resistance to static disassembly* [LD03]. They describe the task of reverse engineering as a two-step process. The first step of reverse engineering consists of the process of disassembly, followed by the decompilation. While disassembly describes the recovery of assembly instructions from machine code to assembly code [LD03] as shown in Figure 3 while decompilation covers the recreation of more complex structures on the basis of previously generated disassembly.

Decompilation is usually not the main purpose of reverse engineering. It aims mostly at getting insight of the internal structure and operating principle of the target application. Recognition of the program internals is not possible without proper disassembly recovery from the binary which is the most basic foundation of reverse engineering applications. To understand the challenges of practical reverse engineering the knowledge of further details related to the compilation process is necessary. In Section 2.1 we present further details about the process of compilation and a short overview about object files.

3 Disassembly

In this section we discuss the two major disassembly techniques and approaches which are widely adopted in academia and industry. Disassembly is an important foundation for binary analysis and advanced operations that base on it. Disassembly Primitives are the basis of disassembly generation and are introduced in Section 3.1. In Section 3.2 the details of linear disassembly are presented. We discuss the recursive disassembly approach in Section 3.3. Recursive disassemblers are the current industry standard although they do not have only advantages over linear disassemblers.

The differences between both approaches are presented in Section 3.4. Different further techniques and approaches as well as possible challenges when dealing with more complex cases are shown in Section 3.5. We focus on static techniques for disassembly generation, also known as *static disassembly*. Besides static approaches there exist also dynamic approaches for disassembly recognition (*dynamic disassembly*) which can include data flow analysis for the prediction of data when processing binaries which are out of scope of our work.

3.1 Disassembly Primitives

There are several aspects mentioned in the literature what disassembly consists of. These different artifacts are called Disassembly Primitives. The quality of the disassembly recovered from binaries is connected to the coverage of the various entities. There are five different artifacts defined in the work of *Andriesse et al.* [ACvdV+16] which are covered by modern disassemblers in industry and research:

- Instructions
- Functions
- Function Signatures
- Callgraph
- Control Flow Graph (CFG)

The introduced primitives are important factors when disassembling applications. *Andriesse* added in his thesis [And17] two disassembly primitives: Basic Blocks (BBs) and Interprocedural CFGs (ICFGs). Both primitives are very common in modern disassemblers, for example IDA Pro uses a basic block notation in its internal application logic.

Instructions are the "mnemonic representation of a machine-level instruction" [And17]. Some instructions alter the control flow of the application and are therefore very important for the detection of basic blocks. There are four different categories of instructions that change the control flow:

- Conditional Jumps
- Unconditional Jumps
- Function Calls
- Return Instructions

Basic Blocks are sequences of instructions directly executed one after another without control flow changes, having exactly one entry and exit. They are typically ended by conditional and unconditional jumps, return instructions, or instructions causing exceptions. Usually a new basic block begins directly after a (conditional) jump and another BB at the target address of the jump.

Function calls move the program counter to another address and push the address of next instruction pointing to the instruction after the call on the stack. Return instructions on the other hand pop the next value from the stack and jump to the value. Usually this is the address which was previously moved to the stack using the call instruction. Both instructions can be used to identify potential function candidates.

Functions are sets of basic blocks which are reachable through paths within the function itself. The function signature is accordingly the formal high level specification consisting of calling convention, number and type of arguments as well as return type. The control flow graph is a structure which connects all basic blocks discovered by the disassembler to a graph structure of one function. Many disassemblers generate one global control flow graph of all functions but *Andriesse* introduces a structure called "Interprocedural Control Flow Graph" (ICFG). The ICFG is a union of the previously generated CFGs. A callgraph is a graph structure that contains all function calls via call instructions. [And17]

3.2 Linear Disassembler

In this section we present the linear approach of generating disassembly from binaries. Linear disassemblers start at a certain address or location of a binary file and interpret the given byte sequence according to their specification of supported instructions, decoding them into a stream of instructions. Tools like *objdump* are typical representatives for the linear disassembly approaches. Algorithm 1 is a pseudo code representation of a possible implementation for linear disassembly approaches. The Linear Sweep algorithm is based on the paper by *Linn and Debray* [LD03].

Algorithm 1 Linear Sweep Algorithm (based on [LD03])

Require: $startAddress$, $endAddress$

1: **procedure** LINEAR_SWEEP(addr)
2: **while** $startAddress \leq addr < endAddress$ **do**
3: $I \leftarrow decode\_instruction\_at\_address\_(addr)$
4: $addr = addr + length(I)$
5: **procedure** MAIN
6: $startAddress \leftarrow$ *address of the first executable byte*
7: $endAddress \leftarrow$ *address of the last executable byte*
8: $linear\_sweep(entry\_point)$

Algorithm 1 shows the linear sweep algorithm described in their publication. The main function is the entry point of the procedure. The start address and the end address are the pre-defined borders for the linear disassembly algorithm and can be e.g. determined by parsing the header of the binary file to detect the text section such as shown in Figure 2 in Section 2.1. The algorithm interprets all bytes between start address and end address as instructions and tries to decode the bytes into the correct assembly instructions. [And17] [LD03]

The concrete output of the linear disassembler and its quality strongly depends on the implementation and on the given input data. There are several exceptions that can occur when parsing byte streams as instructions, i.e. bytes cannot be resolved as instructions when their opcodes are not defined in the ISA of the target architecture (invalid instructions) or large data sections can be interpreted falsely as instructions. Depending on the input data this problem can be reduced, for example when disassembling binary files which have headers pointing to the code section.

Although linear disassemblers can be easily tricked into the generation of invalid instructions or code they are a very simple and therefore very fast approach. The quality of linear disassemblers is also underestimated by researchers in general. *Andriesse et al.* were able to recover 100% of the assembler instructions in ELF binaries via linear disassembly approaches. [ACvdV+16]

3.3 Recursive Disassembler

The recursive disassembly approach is the defacto standard in industry and research solutions [And17]. Tools like IDA Pro analyze binary files and try to obtain the complete control flow graph for every function found in the binary. Algorithm 2 presents the pseudo code representation of a recursive traversal algorithm [LD03].

The recursive disassembler detects functions and new basic blocks by following branches and function calls. This behavior shall help to detect code more accurately and to skip data bytes in the executable instruction sections. [And17]

Algorithm 2 Recursive Traversal Algorithm (based on [LD03])

Require: $startAddress$, $endAddress$
1: **procedure** RECURSIVE_TRAVERSAL(addr)
2: **while** $startAddress \leq addr < endAddress$ **do**
3: **if** *addr* has already been visited **then return**
4: $I \leftarrow decode\_instruction\_at\_address\_(addr)$
5: *mark addr as visited*
6: **if** I is branch or function call **then**
7: **for all** possible targets t of I **do** $recursive\_traversal(t)$
return
8: **else**
9: $addr = addr + length(I)$
10: **procedure** MAIN
11: $startAddress \leftarrow$ *address of the first executable byte*
12: $endAddress \leftarrow$ *address of the last executable byte*
13: $recursive\_traversal(entry\_point)$

The main procedure is the initial entry point of the algorithm. Start and end address are determined correspondingly to the target, for example the begin and end of the text section of a given application. The first call of recursive_traversal can be executed with i.e. the entry point of the binary. [LD03]

For each call of recursive_traversal the algorithm searches sequentially from a given start address to an end address. If an address was already visited by the algorithm then the procedure returns. Else, the algorithm decodes the instruction at the current address and checks the instruction if it is a jump or call instruction. Possible branches and function calls are followed to detect new edges for the control flow graph. The recursive_traversal procedure is called with potential targets as function argument to detect as many parts of code as possible. [LD03]

3.4 Comparison of Linear and Recursive Approaches

The usage of linear and recursive disassemblers have different advantages and disadvantages. While linear disassemblers are known to be more error prone to disassembly errors and the mismatch of data bytes and assembly instructions, they are also very fast due to their simple logic. Recursive disassemblers on the other hand are the defacto standard approach today for disassembly in the industry and

research [And17] [ACvdV+16]. They are more complex and have other issues than linear disassemblers, e.g. they might miss functions that are never called like thread entry points on Windows.

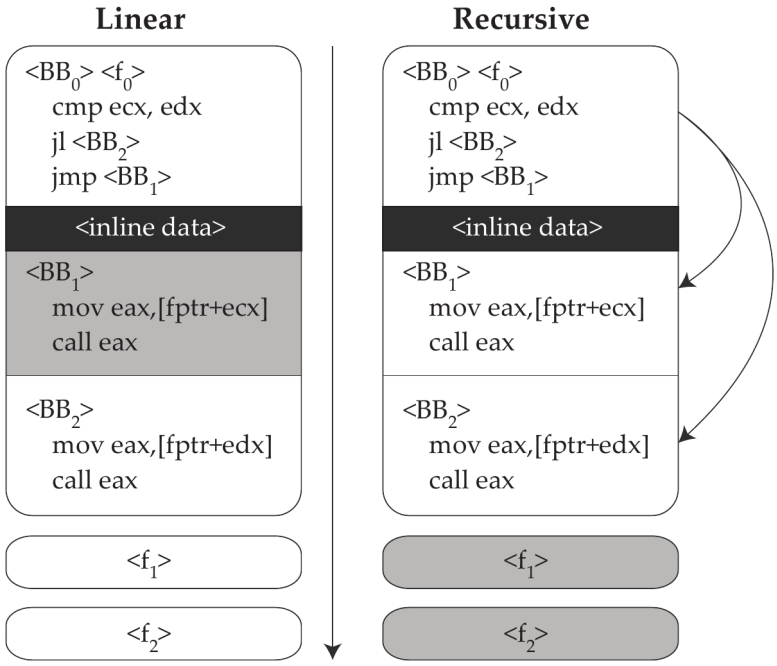

Figure 4: *Linear and recursive disassembly, arrows represent the disassembly flow, uncovered or wrong disassembled code is gray. The illustration is by Andriesse [And17].*

Figure 4 illustrates the difference between linear and recursive disassembly. While linear disassemblers step over all bytes and interpret them as instructions they can be easily confused due to data mixed with code. On the other hand, linear disassemblers can detect functions which are never executed using a jump or call instruction. To decide if a function is executed or not can be hard to distinguish since jumps and calls can be dynamically calculated and are therefore not statically identifiable. This can even lead to a higher amount of recovered functions by choosing a linear disassembly approach than a recursive. [And17] [ACvdV+16] [ASB17]

There are several corner cases discussed in the literature which can result in broken or incomplete disassembly. *Andriesse et. al* presented in their paper "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries" in 2016 that the quality of disassemblers is underestimated by researchers in general [ACvdV+16].

3.5 Challenges of recovering disassembly

Andriesse et al. focused within their publication "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries" also on complex cases that are cited by various researchers as possibly harmful to disassembly and researched their occurrence frequencies [BM11] [MM16] [ACvdV+16]. The following disassembly components are considered hard and complex for proper disassembly generation although they are rare in comparison to most of the code created by compilers:

- Overlapping/Shared Binary Blocks
- Overlapping Instructions
- Inline Data or Jump Tables

- Switches/Case Blocks
- Alignment Bytes
- Multi-Entry Functions
- Tail Calls

Shared binary blocks can be a problem for disassemblers because some implementations try to assign blocks and instructions to only one function for proper disassembly although shared binary blocks are very rare. Overlapping instructions are problematic for disassemblers because it is hard to detect which of the corresponding instruction bytes are the appropriate instruction. When some bytes of the instruction are reused disassemblers often cannot recover the instructions and functions correctly. [ACvdV+16]

Data, jump tables, switches, and case blocks which are inlined between instructions in code sections are hard to detect for disassemblers since they cannot easily distinguish between data and code. Alignment bytes are not necessary executable bytes such as nop instructions and can cause desynchronization from the instruction bytes leading to disassembly errors. Multi-entry functions such as functions which have more than one entry point can complicate the correct function signature recognition. Tails calls are a typical compiler optimization technique that saves the function epilogue such as ret instructions for using e.g. jmp instructions to leave the function. Disassemblers cannot easily identify the end of such functions. The authors discovered that the most complex disassembly primitive to recover are function starts although many of the papers searched by *Andriesse et al.* did not cover these aspect even though this is an important factor when it comes to generating proper disassembly. [ACvdV+16]

Andriesse et al. discovered that most of the time one can easily recover the instructions from legitimate and non-obfuscated applications by using a linear disassembly approach, especially when analyzing ELF files. They compared 30 papers relevant to the topic and showed that the difficulty of the recovery of assembly instructions is generally overrated in the scientific community. While complex cases can be a problem they figured out that most of the time compilers such as GCC and Clang usually do not inline data into the code sections, even not in potentially optimized library code. [ACvdV+16]

4 Related Work

In this section we show subjects and research topics related to disassembly. Disassembly is the first obstacle analysts try to circumvent when reverse engineering applications. High quality disassembly opens up new opportunities for advanced use cases. If the generated disassembly is precise enough one can even reassemble it to new binary files. We introduce advanced topics like binary instrumentation and binary rewriting/reassembling as well as modern dynamic disassembly techniques.

While we focused on static disassembly in our paper there are also different dynamic techniques to leverage the information that are only available during runtime. Dynamic disassembly is a reasearch topic that is not as widely covered in literature as static disassembly [BFM+15] which is quite commonly used in the industry by tools like IDA Pro.

Bonfante et al. showed with their dynamic disassembly prototype *CoDisasm* that it is possible to combine static and dynamic analysis approaches to analyze self-modifying x86 malware. They presented with CoDisasm an automatic binary disassembly framework that is capable to deal with advanced obfuscation techniques like overlapping instructions such as mentioned in Section 3.5.

Another topic related to disassembly is the approach of binary instrumentation. Binary instrumentation describes the process of modifying already compiled binaries to change their behavior for i.e. hardening binaries against possible security flaws [And17]. Again, binary instrumentation can either be statically or dynamically implemented. While static binary instrumentation can be error prone or hard since modification of instructions in binary files might changes pointers and addresses to other program entities. Relocation of all pointers and addresses of programs in general are considered non-trivial and error prone [And17].

Zhang and Sekar presented a first practical approach for automatically hardening COTS (Commercial of the shelf) binaries by applying binary instrumentation to them [ZS13]. They combined a static disassembly approach by using *objdump* and disassembly error detection to obtain the necessary information to implement control flow integrity protection for compiled binaries [ZS13].

Binary rewriting describes the process of reconstruct or patch binary files to alter them in order to e.g. patch vulnerabilities or harden them against possible security flaws [WSB+17]. *Wang et al.* presented with their prototype *Uroboros* an approach to generate valid disassembly that is also reassembleable [WWW15]. The authors reduced their approach to ELF binaries for the x86 and x64 architecture of compiled binaries without any further obfuscation and without any self-modifying code. They were able to disassemble stripped binaries without any debug information precisely enough to reassemble them. *Objdump* and a disassembly validation process were combined to recover the disassembly of the binaries used for their approach.

Wang et al. constructed a new tool called *Ramblr* based on the previous approach by Uroboros [WSB+17]. They evaluated their tool against 106 Linux x86 and x64 applications as well as 143 programs obtained during the Cyber Grand Challenge Qualification. The authors were able to reassemble most of the tested binaries using their prototype Ramblr.

Flores-Montoya and Schulte presented a new technique to rewrite binaries using static disassembly, several heuristics and a Datalog implementation [FS19]. In their contribution they present the tool *Ddisasm* and run extensive tests rewriting several thousands of binaries covering x64 ELF binaries [FS19].

5 Summary & Future Work

Disassembly is a fundamental research topic related to reverse engineering in general but also a key technology when performing sophisticated analyses and operations on binary files such as binary instrumentation and rewriting. In this work we provide an overview about the two primary disassembly approaches and how they work universally. We also present an overview about the current state of research in the context of our topic.

The miscellaneous challenges of recovering disassembly from binaries are still a problem to researchers and analysts although the factor of affecting results by disassembly errors is considered overly pessimistic. *Andriesse et al.* showed that even the unsophisticated method of linear disassembly can restore up to 100% of the instructions for non-trivial real world applications built with GCC. [ACvdV+16]

Function signature detection remains a problem for reverse engineering binary files [ACvdV+16]. *Andriesse et al.* proposed a novel algorithm called Nucleus which shall detect more functions than traditional disassembly approaches. They claim to cover wide parts of the binary by recovering functions with an F-Score of 0.95 [ASB17].

Research of disassembly remains an important topic to improve the tools and frameworks researchers can select. Although various authors were able to present miscellaneous disassembly approaches which were able to even generate disassembly that was could be assembled again, most tools seem still not ready for production compared to industry tools like IDA Pro.

Another prospective challenge are other architectures like ARM or MIPS which are often used in embedded devices. Analysts that try to reverse engineer binaries have to face different challenges. While applications are often deployed in binary formats like ELF that contain explicit information about the internal structure, firmware is usually deployed in customized formats. To disassemble raw binaries or customized firmware, disassemblers need to make assumptions which may lead to imprecise disassembly.

6 Bibliography

[ACvdV+16] 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 Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 583–600. USENIX Association, 2016.

[And17] Dennis Andriesse. *Analyzing and Securing Binaries Through Static Disassembly.* PhD thesis, Vrije Universiteit Amsterdam, June 2017.

[ASB17] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, April 2017.

[BFM+15] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 745–756, New York, NY, USA, 2015. ACM.

[BM11] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[FS19] Antonio Flores-Montoya and Eric M. Schulte. Datalog disassembly. *CoRR*, abs/1906.03969, 2019.

[Kol17] Mitja Kolsek. Did microsoft just manually patch their equation editor executable? why yes, yes they did. (cve-2017-11882). https://blog.0patch.com/2017/11/did-microsoft-just-manually-patch-their.html, 2017. [Online; accessed 04-October-2019].

[LD03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[Lev99] John R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[MM16] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[Sot06] Alexander Sotirov. Hotpatching and the rise of third-party patches. https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sotirov.pdf, 2006. [Online; accessed 04-October-2019].

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems.* Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[WSB+17] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[WWW15] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, Washington, D.C., August 2015. USENIX Association.

[ZS13] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX.