Structural and Differential Analysis for Program Comprehension of Executables

Vom Promotionsausschuss der
Technische Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation

von
Arne Wichmann

aus
Kassel

2017

1. Gutachter: Prof. Dr. Sibylle Schupp
2. Gutachter: Prof. Dr. Heiko Falk

Datum der mündlichen Prüfung: 07.12.2017

Acknowledgements

The present dissertation has been carried out at the Institute for Software Systems.
I thank Prof. Dr. Sibylle Schupp for her support of this work, her criticism of the manuscript, and many constructive discussions.
I thank Andreas Dierks, Dr. Gerko Wende, and many others at Lufthansa Technik, who supported my research.
I thank Prof. Dr. Heiko Falk for his support and criticism of this work.
Special thanks goes to Sven Mattsen, with whom I shared many fun days at the office.
Additionally, I thank the students, whose theses I supervised, for allowing views into various areas of research. I thank all of the institute's members for their work, collaborations, and all discussions. And I thank all who have helped me on my way to completion of this thesis.

Especially, I thank my mother, Dagmar, for her believe in my work.

Most importantly, I thank my wife, Gesche, and my children, Anton and Ida. For everything.

Contents

1 Introduction 1

2 On Binaries 7
2.1 Code Generation Toolchain 7
2.2 Disassembler 8
2.3 The Executable 11
2.4 Order-Preserving Variations 13

3 Program Comprehension 15
3.1 General Program Comprehension 15
3.2 Patterns and the Cognitive Limit 18
3.3 Structural Analyses 20
3.4 Differential Analyses 22
3.4.1 Source Code 22
3.4.2 Executables 27

4 Ordered Control-Flow Graph 29
4.1 Requirements 29
4.2 Formal Description 30
4.3 Fully Worked Example 34
4.4 Graphical View of Ordered Control-Flow Graphs 39

5 Structural Analysis for Program Comprehension 45
5.1 Pattern Definitions 49
5.1.1 Function Patterns 50
5.1.2 Module Patterns 52
5.2 Pattern Examples 55
5.2.1 Hosted Executables 55
5.2.2 Runtime Library 62
5.2.3 Stand-Alone Executable 63
5.3 Case Study 66

6 Differential Analysis 73
6.1 Function Size Sequence Alignment 73
6.2 Formal Description 77

6.3 Function Size Similarities 79
6.4 Fully Worked Examples 84
6.5 Quality Estimates 90

7 Evaluation 93

7.1 Setup 93
7.1.1 Ground Truth 94
7.1.2 Test Suite Construction 99
7.1.3 Empirical Selection of Similarity 102
7.1.4 Canonical Case Study 104
7.2 Evaluation Data 108
7.2.1 Product Lines 108
7.2.2 Dissimilar Executables 132
7.2.3 Validation 137
7.3 Summary 147

8 Conclusion 157

Bibliography 161

1 Introduction

An executable is a file that contains the information necessary for a specific computational task to be performed by a machine. This information usually comprises the instructions to be executed by a processor and the data necessary to perform the task. The definitions of executables range widely in the abstraction level of the contained code and additional information included. The code abstraction levels can range from scripting language files, via high-level bytecode, to instructions in a raw memory image. Similarly, the additional data can be original source code, links to source code and names, intermediate representations, and section information. In this thesis, we consider executables that contain instructions on machine or bytecode level without additional information.

The application scenario of this work are analyses for reverse engineering, performed by a human analyst on one or multiple unknown executables. The executables are extracted from an embedded system. If there are multiple executables these originate from the same or very similar systems. The cause for the analyses are questions about the behaviour of the software. Specifically, one can imagine an analyst investigating a product line of pocket calculators that exhibit a miscalculation in a specific case. In such a case for an analyst external to the production company, access to the source code used to program the calculators is not possible. Instead, the executables are inspected to locate the origin of the miscalculation. In a single executable analysis it is important to identify the parts of the software that are to be analysed efficiently. If there are multiple executables from calculators, where some executables show the behaviour and others do not, an efficient way of transferring the information from one executable to another and especially a way to identify the differences between the executables are necessary.

The challenges of analysing executables range from reconstructing obfuscated control flow to plain comprehension of the software presented at a concrete level. While some of the tasks are supported by software, much of the analysis time is spent on manually comprehending the software. Nonetheless, in many cases, much of the necessary information can be derived by analysing the executable structure (sections, libraries, etc.) and included entity names. But, in the context of reverse engineering software

from embedded systems, the worst case has to be assumed, which is a completely unstructured, raw executable without any names for control- or data-entities.

Program comprehension is a cognitive task of humans. It is usually necessary whenever software needs to be created, modified, or described in another format. Generally, the necessary effort for this task is considered to be reduced by the availability of any kind of documentation, descriptive names in the source code, and the chosen abstractions and clusterings in the software such as functions, modules, or libraries. Program comprehension on binaries is challenging because none of such helpful information can be assumed to be available in the general case. Additionally, our executables contain information at a very low and very verbose level, which challenges the cognitive abilities of a human. Typically, these challenging aspects are caused by the removal of the former abstractions on source-code level, like lost modularisations or expansions of simple high-level language statements to multiple instructions on machine level.

In this thesis, we set out to aid a human performing a program comprehension task by introducing a structural analysis that helps to regain some of the abstractions from the source-code level lost in the executable. Currently, such an analysis has to be performed manually by scanning and reading the code and recognising the roles of functions in the general control structure of a program. Such an analysis is very time consuming and any aid that reduces or prioritises the amount of information is very helpful, as often the information necessary to comprehend the program and to answer a high-level question about the program only depends on small parts of the software. In this case, locating these parts easily becomes the most time-consuming part of the analysis.
To provide such aid, we start with disassembled code at function level and represent the control coupling in and between functions, and their positions in the executable in a new, order-sensitive representation. We try to find key spots in the executable, as well as relations between functions or groups of functions using recognisable patterns in a graphical representation.

In a second step, we provide a differential analysis for executables on the new order-sensitive representation. In many cases of analyses, not only one but several similar executables are available. In case of malware, these may be an untainted and a contaminated build of the same software, or in

case of embedded systems, these may be different versions (with different features) of the same system. An analyst can perform an analysis on a second executable much faster if the information known about, or reverse-engineered from, the first executable can be transferred automatically to the new executable. Additionally, in a parallel analysis of two executables, information can be gained from recognising the identical and varying parts between the executables. To provide such a differential analysis, we use a function-level alignment of pairs of our representation that employs similarity measures on pairs of functions.

Program comprehension is a cognitive task of humans. It is usually While our structural and differential analyses are applicable to most executables, they only depend on very basic information. This allows their application to executables from legacy embedded systems. On such systems, the typically available artefact is a raw memory image, obtained directly from a chip, without any links to the original source code (names, line numbers), or access to any documentation or source code.

As base for our structural and differential analysis, we introduce an ordered control-flow graph (OCFG), which extends a traditional control-flow graph (CFG) by including an order of the entries derived from the order of functions as they appear in an executable. For the structural analysis, this order contains information from the compilation process of the executable, which is hidden in the traditional CFG, but is usable for our analyses. For the differential analysis, we make the observation that the order of appearance of functions in the executable is stable against several variations, and can be exploited to create an alignment of two function sequences.

We evaluate the patterns from our structural analysis by testing them on a set of various executables, inspecting each match and performing an extended case study for one executable. The evaluation of the differential analysis is performed on a set of similar executables that represent executables from embedded software product lines, and on a set of dissimilar executables. Additionally, the results are validated on two sets of executables obtained from Stojanovic, Radivojevic, and Cvetanovic [40]. For each of these evaluation parts, we inspect whether the sequence of functions in the executable can be assumed as stable, and then continue to apply our differential analysis and inspect its result in terms of actual alignment

quality and estimated alignment quality. Additionally, we inspect the effectiveness of our alignment by comparing the achieved result with the similarity potential.
In summary, we make the following contributions:

- A new, order-sensitive representation of intra- and interprocedural control dependences called OCFG, along with one view for inspection of single executables and one view for comparison of two executables. We provide a definition and a graphical representation using scatter plots.

- A new way of abstracting from control dependences at statement level to control dependences between pairs of functions or sets of functions ("modules"). We introduce nine visual patterns, which exploit different properties of the OCFG. We apply these patterns to four executables selected from standard benchmarks, a library, and a standalone kernel and inspect the results in a total of seventeen occurrences.

- An algorithm for identifying the similarities in pairs of OCFGs, based on function-level alignment. The algorithm is parametrised by similarity measures based on function sizes that tolerate to different degrees order-preserving modifications of either of the two OCFGs. We provide an empirically validated measure that estimates the quality of the alignment.

- An extended evaluation of the differential analysis in terms of potential, estimated, and actual alignment quality on a set of executables representing an embedded software product line, and a set of differing executables. These sets are split into 5 cases, containing 18, 68, 28, 63, and 20 executables each, which results in 5183 pairs. We calculate an alignment for each of our 10 parametrisations and for each of these pairs, and inspect the best performing parametrisation. To obtain ground truth about the sequence similarity and similarity, we provide a similarity measure using function names. We validate the differential analysis evaluation on executables from Stojanovic, Radivojevic, and Cvetanovic [40]. These are split into two sets of 50 and 14 executables, or 1380 pairs, which are processed as above.

- An implementation of the structural and differential analysis called Küstennebel that uses data from IDA Pro[1] and produces the plots and alignments is available online.[2]

Note that, although we aim to aid a human analyst, our analyses are evaluated at a technical level, but not in the cognitive context.

Outline Chapter 1 introduces the thesis. The background and related work are presented in Chapters 2 and 3. Chapter 2 introduces the code generation toolchain, disassemblers, executable formats, and order-preserving variations to prepare the data necessary for the definition of the OCFG. Chapter 3 introduces program comprehension in general, discusses the human cognitive limit, and the usage of patterns as a way to evade this limit. Additionally, it introduces structural and differential analyses on executables in the context of program comprehension. Lastly, as part of the discussion of differential analyses and to prepare our differential analysis, it introduces basic sequence alignment algorithms.

Our definitions and analyses are presented in Chapters 4 to 6. In Chapter 4 we define the OCFG, provide a fully worked example, and define one view of the OCFG for single executables and one view for multiple executables. Chapter 5 describes our structural analysis (SA). It discusses properties of several structural hotspots for program comprehension and defines patterns to graphically identify such hotspots. Additionally, we apply the patterns to a set of executables and inspect the results. The chapter concludes with a case study on an embedded executable. Chapter 6 describes our differential analysis (DA). It defines a sequence alignment parametrised by function-size based comparison functions and introduces measures that estimate the alignment quality.

Chapter 7 is the large evaluation chapter concerned with function sequences, split into three sections. Section 7.1 discusses the acquisition of ground truth similarity data and the construction of the test suite, and defines a canonical description of case studies for the evaluation of the alignment. Section 7.2 contains the data of case studies for similar executables

[1] *IDA Pro: Interactive Disassembler*. URL: `http://www.hex-rays.com/products/ida/index.shtml` (visited on 05/19/2017).

[2] *Küstennebel*. URL: `https://github.com/arnew/kuestennebel` (visited on 05/19/2017).

from several product lines, different executables, and a validation study against externally provided executables. Section 7.3 summarises the case studies and evaluates the different quality aspects.
Chapter 8 concludes the thesis and gives an outlook on future work.

2 On Binaries

The ordered control-flow graph (OCFG) that we introduce in Chapter 4 uses data from executables. In this chapter, we discuss the code generation toolchain, the common denominator between several disassemblers, assembled code in general, and several executable types, to introduce the minimal input data for the OCFG. Additionally, to prepare our differential analysis (see Chapter 6), we introduce similar executables, which are executables built from a common set of assets with variations that preserve the order of functions in the executable.

2.1 Code Generation Toolchain

This section introcudes the traditional code generation toolchain. A traditional compilation scheme allows the development of software in separate source files, and modular compilation of the software. Compilers such as GCC or Clang usually compile each source-code file into an object file, which is then linked into an executable. Figure 2.1 shows the Compilation Toolchain as it is used by the example presented later in Section 4.3. Here, three source files (`main.c`, `b.c`, and `a.c`) are individually passed to the compiler and compiled into separate object files (`main.o`, `b.o`, and `a.o`). These are linked into a final executable (`main`). In traditional compilation, part of the original structure is preserved by the modular nature of the compilation. We inspect this behaviour later throughout this thesis.

Modern compilation techniques can process the source code in a way, where whole program optimisations becomes a possibility, such as, for example,

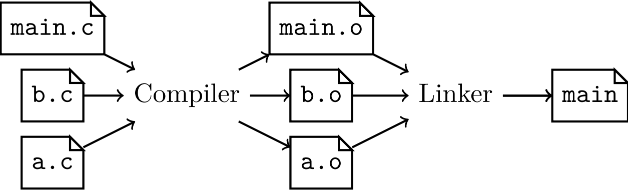


Figure 2.1: Example of Elements in Traditional Code Generation Chain

dead-code elimination or cache page reorganisation. But, currently, these techniques are not widely deployed, and so far remain a speciality for, for example, performance critical software, where an optimisation of the code to fit ideally into cache pages is important. As we will later see, such behaviour potentially disrupts the stability of the order of functions in the executable. Nonetheless, there is hope in cases like these, as these compilers also aim at reproducible builds as well as a stable optimisation (no small changes cause a reorganisation of large parts of the software).

A simplified view on the order of the code produced by the compilation of software is the following. Each function consists of several basic blocks, which are chunks of instructions that get executed sequentially, may be jumped to or called from several positions, and may continue at several positions. These basic blocks are the nodes in the control flow graph, and jumps and calls are the edges. During code generation the control flow graph is simply traversed using traversal scheme like breadth-first search. For traditional compilation, the code generated for each function of a source file is concatenated to form an object file. The linker then concatenates the object files into an executabe.

For the modern compilation techniques, the traversal schemes can be considerably more complex. But, as an examination of a simple example of link-time optimised software at the end of the thesis in Section 7.3 shows, even these schemes can produce a stable function sequence.

2.2 Disassembler

A disassembler is a software that transforms machine language instructions contained in an executable into assembly language. As a by-product, many disassemblers also produce a (sometimes approximative) control-flow graph of the software.

Instructions in assembly are typically represented using mnemonics that allow operations to be identified, and a possibly empty lists of operands. Assembly language instructions are translated to their machine language counterpart during the build process of the software. Except for esoteric corner cases, each machine language instruction can be associated with an address where it is stored when the program is executed. Each instruction is encoded with a certain length that is fixed on some architectures and variable

on others. These addresses are used in a program counter in the processor to realize the program's control flow. In some cases, part of the address is calculated by the use of segment or base registers, but disassemblers commonly map the program into a single unified address space and provide abstractions to map the targets of jumps and calls onto this representation.

There are two basic control-flow abstractions used at the instruction level. The first is the jump instruction `jmp 0x1234` (set program counter to 0x1234), which, sometimes based on a condition `jnz 0x1234` (set program counter to 0x1234 if the result of the last arithmetic operation was not zero), changes the value of the program counter directly. The second is the call instruction `call 0x1234` (call function at 0x1234), which also changes the program counter value but stores the value of the unmodified program counter on a stack. Together with the call, there is the return instruction `ret`, which allows the program to restore this original value and to continue execution after the position of the call. Using these two control mechanisms, most software abstracts its code into functions. The control flow inside a function (intraprocedural flow) is then modelled using jumps, and the control flow between functions (interprocedural flow) using call and return instructions.

To decode instructions after loading an executable file a disassembler has to decide which addresses are associated with instructions. The most basic disassembling strategy is a linear sweep disassembling. This strategy starts at a given address and decodes instructions based on the instruction lengths. This strategy has limitations when data is interleaved with the instructions, as it may be falsely decoded. On architectures with unaligned or variable length instructions, false decoding may additionally lead to misaligned decoding of instructions (although they usually realign within a few decoded instructions [33]). Nonetheless this strategy is very useful for high-level executables in the Executable and Linking Format (ELF), where separation information between code and data is available, and the entry points to functions are given. The most prominent linear sweep disassembler is objdump contained in the Binutils[1] related to the GNU's not unix (GNU) compiler collection.

[1] *GNU Binutils.* URL: `https://www.gnu.org/software/binutils/` (visited on 05/19/2017).

The workhorse disassemblers of the reverse-engineering community, namely IDA Pro[2] and Radare2[3], use a recursive-descent strategy to disassembling. Again, these start from a set of entry points. In contrast to the linear sweep disassembly decoding of the instructions, they do not assume that instructions form a continuous chain in memory, but model a simple representation of the instruction control-flow behaviour. That means, they approximate jump target addresses and continue decoding instructions at their targets (maintaining a queue for multiple targets), approximate call target addresses and queue them as new entry points of functions, and stop sweep decoding of instructions after halt, return, or unconditional jumps. Typically, recursive descent disassemblers can calculate an exact target for many of the target addresses, as those are often encoded as absolute or relative addresses in the machine code, but in case of indirect jumps (values stored in registers), they usually fall short, except if they have heuristics to approximate the register value. In case of such problems, these tools allow a user to interactively provide new entry points and to explicitly specify the targets of such indirect jumps and calls. A recent approach to automatically create heuristics and reduce the workload for the user by Bao, Burket, Woo, Turner, and Brumley [3] employs a learning system to recognise the starts and boundaries of functions.

The most complex disassembler strategy is used by tools like the binary analysis platform (BAP)[4] [6] or Jakstab[5] [17] and BDDstab[6] [23, 24]. They use very detailed models of the program's instructions and can calculate approximations of the program's control flow by tracking most of the program's data flow. This allows approximation of target addresses even in the case of complex calculations performed on the registers and memory used as an indirect jump or call target. But, disassembly built upon such complex analyses is still in the early stages of development and computationally expensive.

[2] *IDA Pro: Interactive Disassembler.* URL: `http://www.hex-rays.com/products/ida/index.shtml` (visited on 05/19/2017).

[3] *Radare Reverse Engineering Framework.* URL: `https://radare.org/r/` (visited on 05/19/2017).

[4] *Binary Analysis Platform.* URL: `https://github.com/BinaryAnalysisPlatform/bap` (visited on 05/19/2017).

[5] *The Jakstab Static Analysis Platform for Binaries.* URL: `http://www.jakstab.org` (visited on 05/19/2017).

[6] *BDDStab.* URL: `https://www.tuhh.de/sts/research/projects/bddstab.html` (visited on 05/19/2017).

Table 2.1: Features of Major Executable Standards

Type	File Extension	Addresses	Debug	Linking
ELF		Per Segment	Yes	Yes
PE	`.exe`, `.dll`	Per Section	Yes	Yes
COFF		Per Section	Yes	Yes
a.out		Per Section	Yes	Yes
COM	`.com`	Fixed	No	No
srec	`.s9`, `.srec`	Flexible	No	No
hex	`.hex`, `.ihex`	Flexible	No	No
bin	`.bin`	Unknown	No	No

To summarise the properties of disassemblers for our structural and differential analyses, and for the construction of the ordered control-flow graph (OCFG), all disassemblers decode instructions at specific addresses. Additionally, the disassemblers provide control-flow information on different levels. The lowest level is the resolution of targets for immediate jumps and calls, as well as grouping of code into basic blocks.
For our analyses, we assume that code is grouped into basic blocks and for each block, a list of possible next blocks is given. Additionally, the targets for function calls can be extracted. As we will see later, an unsound approximation (mostly with respect to indirect jumps or calls) is sufficient for our analyses.

2.3 The Executable

Executables exist in a multitude of file formats. In this thesis, we consider the major types of executables standards listed in Table 2.1. Besides their representation in the file, the executables mostly differ in three categories: Their handling of addresses and correspondingly their segment and section model; the inclusion of debug information like relation of instructions to source code positions and names of functions and data; and finally, their ability to be linked in as a library or to link to libraries themselves at compile time or at run time.

Executables with a complex file representation like the Executable and Linking Format (ELF), Portable Executable (PE), Common Object File Format (COFF), or a.out types can contain all of the abovementioned information. That means, they can contain several code segments or sections, which are to be placed at different base addresses in the unified program address space, can provide debug information, and support linking. It is possible that, even if at compile time all of this information is given, some of the information gets removed during a stripping of the binary. In this case, usually only the necessary information for running and linking the software is left in the executable. Often, when such executables are supposed to be linked, some of the names of functions are preserved, as they are necessary to identify the public interface of the contained software to be linked against.

The next group of executables are the COM and `srec` or `hex` executables. These normally do not contain debug information and are not linkable in the general sense. Addresses of the contained code are either fixed by convention (COM) or explicitly given for all of the contained data (`srec` or `hex`).

The baseline type of executables is the raw binary (`bin`). This is the file representation of what is often contained directly in the program memory chips of embedded systems. Addresses are no longer explicitly present and often have to be inferred from absolute accesses in the program or from the wiring of the memory chips.

In the analysis of malware, obfuscations (from “obscure” and “confuse”) play an important role. These often introduce wrong information in the debug information and try to circumvent specific analysis techniques. In the domain of embedded executables, where the author performed analyses in an industrial context, no obfuscations were observed at all.

While the executables allow different address spaces, for our analyses, we take the disassembled data from IDA Pro, which unifies the code into a single address space. Generally, this is no problem, as the unified space can be chosen arbitrarily large, so collisions can be avoided.

2.4 Order-Preserving Variations

An important aspect for the differential analyses are executables that are similar. We consider two executables as similar, if they contain the same functions, and roughly perform the same functionality. Commonly, for our analyses, such similar executables are obtained by sampling the software of multiple systems or at multiple times.

A more formal approach of such similar software is a software product line, which is “a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.”[7]

With this definition in mind, we assume that the variations in the software are mostly introduced by normal software development, such as adding new features, fixing bugs, porting the software, and rebuilding it. These activities imply variation of the source code and of the build process of the software.

Our differential analysis exploits the observation that, although these variations imply changes of the functions in the executable, the overall sequence of functions stays stable. Next, we inspect several of the order-preserving variations.

The primary and most prevalent variation is the versioning or revision of the software. On the executable level, this variation introduces, removes, and modifies some functions. The next variation are changes in the optimisation of the software. These can be caused by setting different options during the build process, or simply by changes in the code that enable certain optimisations. Typically, optimisations are grouped by their scope, that is, peephole optimisations local to few instructions, local optimisations inside of a basic block, loop optimisations for loops in functions, global optimisations inside of functions (intraprocedural), and whole-program optimisations (interprocedural). Except for whole-program optimisations, all of these imply changes in the size of their corresponding functions. Examples are saving code in peephole and local optimisations, introducing code in loop unrolling, or eliminating dead branches. Whole-program optimisations

[7] Software Engineering Institute, Carnegie Mellon University. *Software Product Lines.* URL: `http://www.sei.cmu.edu/productlines/` (visited on 05/19/2017).

introduce the possibility for functions to be removed (dead-code elimination or inlining).

Another aspect of variations especially for embedded systems are changes in the platform or architecture. Changes in platform or architecture imply changes in the set of included driver modules, but can also include changes in specific instruction set extensions, which may or may not be available. A change in the modules mostly consists of added or removed functions. Different instruction set extensions (for example, availability of a floating point unit) may necessitate the inclusion of a software floating point library and a change of all floating point instructions to calls to such a library. Architecture changes imply a complete change of the instruction set and its available features, as well as a possible change in the endianness. Changes in the instruction set, again, imply changes in the function sizes.

Lastly, the whole source tree can be compiled with another compiler. Such a compiler change usually implies a change in the applied optimisations and code generation. On the executable level these compiler changes can lead to addition, deletion, and modifications to all of the functions.

An important aspect is that most, if not all of the above modifications preserve the order of functions in the executable. The general observation is that the embedded software is commonly compiled from several separate source files to several object files, which are then linked together to form the executable. In most build processes, the order of these source files is predetermined and there is no necessity for compilers to change the order of functions in an object file. This observation implies that with traditional compilation schemes there is a good chance that the order of functions in and among the source files is transferred into the executable. In summary, the address values order the functions and control flow in the executable and this order is stable to a certain degree across the variations in software product lines.

The algorithms and evaluation in this thesis are designed to perform best on executables with order-preserving variations. For the structural analysis, the similarity in the software implies a similarity in our later defined representation and its views. For the differential analysis, the order-preserving variations allow the application of a sequence alignment, although the parametrisation of the algorithm still needs to be tuned to handle such variations.

3 Program Comprehension

In this chapter, we discuss program comprehension in general, discuss which artefacts used for program comprehension (PC) are available in our application, and detail how processing these artefacts is limited by human cognition. We introduce patterns as remedy against the cognitive limit. We also discuss related structural analyses on executables, as well as differential analyses for source code and executables. Additionally, to prepare our differential analysis, we introduce alignment algorithms used to calculate differences between various types of sequences.

We inspect the program comprehension process to understand the special challenges when reverse engineering executables. Later, we provide analyses that aim to aid a human analyst, which are successful in a technical evaluation, but are not evaluated in the cognitive context.

3.1 General Program Comprehension

Program Comprehension (PC) is a wide field. Janet Siegmund summarised the past, present, and future of program comprehension [37]. Apart from being split into past, present and future, her article mentions four major divisions of program comprehension research, namely measuring PC, modelling PC, programming languages, and programming tools. We follow this separation in our discussion of general program comprehension and summarise her work.

Program comprehension needs to reflect on the level of comprehension a programmer has achieved. While it is currently impossible to read the state of a programmer's mind directly, several alternative approaches are used, like observing programmers, questioning memorisation, assessing with tests or comprehension tasks, and even neuroimaging. Through simple observation of a developer's activities, optionally enriched by the developer thinking out loud, the process of program comprehension can be studied. Historically, programmers had been challenged with recalling and writing down as much of a program as possible as a measure of their level of comprehension. Today, the sizes of programs have grown beyond any memorisation capability. By

Table 3.1: Common Elements of Cognitive Models

Category	Elements	
Knowledge	General, Independent vs. Appl. Specific	
Mental Model	Static	Text Structures, Chunks, Plans, Hypotheses, Beacons;
	Dynamic	Strategies, Actions, Episodes, Processes
Facilitating Knowledge Acquisition	Beacons, Coding Standards, Data Structures, Mnemonic Naming, Programming Plans	
Expert Characteristics	Knowledge, Schemas, Flexibility	

Summarised from: Mayrhauser and Vans [25].

examining the developer with specific tests or comprehension tasks the level of comprehension a developer has achieved, can be evaluated.
A future development mentioned in Siegmund [37] is the development of neuroimaging technologies that allow a more direct inspection of a developer's brain activities.

Various models of program comprehension processes have been developed over the years. The most common model types are top-down, bottom-up, and mixed models. In top-down models, the analysis of a program starts at the most abstract components, like frameworks, modules, classes, and then proceeds down to functions and individual instructions. In contrast to top-down, for bottom-up models the analysis starts at individual instructions and proceeds to abstract components. Since developers rarely comply with such simple models, mixed models exist that combine both the top-down and bottom-up approaches. Siegmund [37] writes about such mixed approaches that "[...] programmers use top-down comprehension where possible and bottom-up comprehension only when necessary, because top-down comprehension is more efficient than examining the code statement by statement." She cites Shaft and Vessey [35] writing about the relevance of application domain knowledge as source of this observation.

Table 3.1 shows the common elements of cognitive models, summarised from Mayrhauser and Vans [25]. The work states four major categories of elements, namely knowledge, the mental model, facilitating knowledge acquisition, and expert characteristics. In each category, elements of the cognitive models are described. For example, programmers use their general, independent or application specific knowledge to provide a context for their mental model. The mental model then consists of static and dynamic elements of the software itself. These can for example be the textual structure that the software is written in, or a strategy like divide-and-conquer, which can be easily recognised and integrated into the mental model of the software. Other knowledge can be acquired from elements implicitly encoded into the program, like certain beacons (e.g., `printf`), recognisable data structures, mnemonic names used in the software, or plans of the software in the documentation. Lastly, several characteristics of the expert also play a role in the construction of the cognitive model of the software.

The above observation, together with the list of common elements, nicely describes the problems with program comprehension of executables, where most, if not all, of the abstract information enabling top-down comprehension is lost, and the analysis therefore has to be performed on instructions and functions. In Section 3.2, we discuss more details on the effects of missing information during structural analyses.

In the past, program comprehension developed alongside programming languages from assembly level, via high-level languages, eventually to modern application programming interface (API) driven development and domain specific languages (DSLs).

Concerns derived from program comprehension have driven the development of modern integrated development environments (IDEs) that allow interactively browsing of the code as well as advanced presentations of the code using indentation or colour coding.

For the future, Janet Siegmund describes ideas of monitoring the programmer and providing aids in situations when a programmer reaches the human cognitive limit (see [27]).

Table 3.2: Cues to Understanding a Program

(a)

Internal to the Program Text.	
1.	Prologue comments, including data and variable dictionaries.
2.	Variable, structure, procedure and label names.
3.	Declarations of data divisions.
4.	Interline comments.
5.	Indentation or pretty-printing.
6.	Subroutine or module structure.
7.	I/O Formats, headers, and device or channel assignments.
8.	Action of statements, including organisation.

(b)

External.	
1.	User's manuals.
2.	Program logic manuals.
3.	Flowcharts.
4.	Cross reference listings.
5.	Published descriptions of algorithms or techniques.

From Brooks [5]

3.2 Patterns and the Cognitive Limit

As mentioned in the discussion of programming tools in the last section, humans have a cognitive limit [27]: Humans have the capability to remember about 7±2 random items. If a human has to process more items, then these have to be somehow summarised or the items need to be in some kind of sequence.

Table 3.2 summarises what Brooks [5] writes about the cues to understanding a program. The table is split into two parts, which name the cues internal and external to the program text. In the worst case of executable analysis from embedded systems, none of the external cues are present. From the internal cues, all of the information in the source code needs to be assumed lost (numbers 1 to 5). Parts of the information of the subrou-

tine and module structure (6) are still explicitly or implicitly encoded in the executable, and our structural analysis is targeted to it. The detailed semantics of the operation of the system (7 and 8) is still encoded in the executable, although function names are usually stripped, and can be used to reconstruct the system's behaviour from bottom up, when data sheets for the components are available.

The major problem of such a bottom-up analysis is that due to the verbosity of assembly language, code reconstruction is only feasible for the most important parts of the program to analyse. But without the high-level information, there are generally no clues as to which of the executable's parts are important. The structural analysis reconstructs some of this information to allow assessment, understanding, or modification of the programs parts.

Clusters and Patterns Biggerstaff, Mitbander, and Webster [4] describe how to obtain information about a program's concepts using four scenarios, namely suggestive data- and function names (1 and 2), patterns of relationships (3), and intelligent agents (4). For our analysis, the scenarios 1 and 2 are infeasible, since the executable's names are stripped, but scenario 3 gives a clue about the intention of our structural analysis:

> "Another approach to program analysis is to try to identify the clusters of functions and data that appear to be closely related in order to form a structural framework on which to hang the details of the program. We call these clusters modules, not to be confused with files, objects, or other formal programming language structures."[4]

Our structural analysis aids a human analyst to reconstruct such clusters or modules, by presenting the control-flow information in an appropriate way. To further reduce the cognitive complexity, we follow the ideas presented by Gamma [11] and Lilienthal [21] about patterns in software development and analysis and provide several patterns that apply to the control-flow information, and represent higher-level constructs in the software.

The use of clustering to modularise a software is also discussed by Lethbridge and Anquetil [19], but this work concentrates on applications for source code and the general abstract concept.

3.3 Structural Analyses

In this section, we first look at structural analyses that transform the code between different programming language levels, like decompilers, or source to source transformation systems. Next, we describe several approaches that analyse the structure of executables using visualisation techniques. Finally, we describe a formal structural analysis that groups parts of a software into concepts, and is similar to our structural analysis using patterns, but less visually intuitive.

A major effort in structural analysis for executables on the intraprocedural level is the development of decompilers. Historically, Cifuentes [8], Ramsey and Fernandez [31], and Van Emmerik [41] have developed such decompilers using lifting of assembly to an intermediate language and transforming the code to the C language. Currently, commercially available decompilers are, for example, IDA Pro[1], Hopper[2], and RetDec[3]. Other code inspection tools that use lifting techniques are CodeSurfer/X86[4] [2] and BinNavi[5].

A criticism on the use of decompilers for program comprehension is that while decompilers transform the syntax and semantics of the assembly code into a higher level language (usually C), they do not necessarily increase the level of abstraction. Sometimes, if the decompiler is finely tuned (usually through some heuristics) to the original compiler, a programmer can understand the produced program. But generally, and especially in the case of embedded system compilers, the decompilers do not possess these heuristics and the transformation retains the verbosity (low-level noise) of the assembly language.
A simple example is the transformation of memory moves from one address to another (`mov [0x1234], [0x2345]`) to an equivalent construct using explicit pointers in the C language (`*((int*)0x2345) = *((int*)0x1234)`).

[1] *IDA Pro: Interactive Disassembler.* URL: `http://www.hex-rays.com/products/ida/index.shtml` (visited on 05/19/2017).
[2] *Hopper: The macOS and Linux Disassembler.* URL: `https://www.hopperapp.com` (visited on 05/19/2017).
[3] *retdec: Retargetable Decompiler.* URL: `https://retdec.com` (visited on 05/19/2017).
[4] *CodeSurfer/x86.* URL: `https://www.grammatech.com/products/codesurfer` (visited on 05/19/2017).
[5] zynamics. *BinNavi.* URL: `https://www.zynamics.com/binnavi.html` (visited on 05/19/2017).

Although the C language statement is semantically equivalent, none of the previous high-level information (like declaration of the variable or the data types) are necessarily reconstructed. Apart from such inconveniences for the data-flow, the control-flow output of decompilers currently still tries to be more idiomatic (e.g., avoiding `goto`) [48].

Additionally to these specialised decompilers, but usually for a program maintenance tasks, it is possible to use source-to-source transformation frameworks like TXL[6] [9] or FermaT[7] on assembly language code to create application-specific transformations. For example, Ward [43] uses the FermaT framework to transfer assembler to C. But again, these solutions are highly individual and cannot be applied generally for our application.

There are a few approaches to inspecting executables by using their byte values as grey values in images (see BinID2[8] or Nataraj, Karthikeyan, Jacob, and Manjunath [28]). Additionally, there have been efforts to calculate and derive information from the entropy in the byte values of the executables (see Han, Kang, and Im [13]). These analyses allow a fast identification of the data and program sections contained in an executable and are especially useful to detect hidden code sections or aid in the analysis of unknown or obscure executable formats.

Generally, visualisations of software often use box and line or arrow types of visualisation (see Linos, Aubet, Dumas, Helleboid, Lejeune, and Tulula [22] for an early overview). This type is very insightful for smaller parts of the software (limited number of elements). But, to be applicable to larger parts of the software either very good graph layouts or special summarising techniques have to be used.

Similar to our representation approach, adjacency matrices and corresponding scatter plots are used to cope with the large number of couplings between parts of a software, for example, for the analysis of dependencies in the .Net framework (see Lämmel, Linke, Pek, and Varanovich [18]).

[6] *The TXL Programming Language.* URL: `https://www.txl.ca/` (visited on 05/19/2017).
[7] *FermaT: The FermaT Program Transformation System.* URL: `http://www.cse.dmu.ac.uk/~mward/fermat.html` (visited on 05/19/2017).
[8] *BinID2.* URL: `http://www.phenoelit.org/BinID/index.html` (visited on 05/19/2017).

Another structural analysis is the formal concept analysis by Ganter, Stumme, and Wille [12]. Similar to the patterns in our structural analysis, here, entities are grouped into a partially ordered set (lattice) of concepts based on the presence of common attributes. Ignoring the address-based order in the control flow, as we use for our analyses, it could be possible to use formal concept analysis to perform an analysis similar to our structural analysis but with much higher computational cost, and with less visually intuitive results.

In Chapter 5, after the introduction of the ordered control-flow graph (OCFG), we introduce visual patterns that aid a human analyst to perform structural analyses on executables in an intuitive way and that limits the amount of information to be remembered simultaneously and evade the limits set by human cognition.

3.4 Differential Analyses

In this section, we discuss several approaches to differential analyses on source code and assembly level. For our differential analysis introduced in Chapter 6, we also take an excursion to algorithms for subsequence alignment.

3.4.1 Source Code

Probably the highest number of differential analyses on source code is performed in version control systems, where the changes to the system need to be stored efficiently as well as presentable to a human. Additionally, many software systems contain duplications of source code in themselves, called code clones. Roy and Cordy [34] summarise much of the work in the field, which has used a multitude of different techniques and can be used in many applications. For our differential analysis, the more advanced techniques are not in scope, as currently these techniques are relatively time-consuming and the results do not seem to be worth the trade-off. Instead, our differential analysis produces a function-level result with the techniques traditionally associated with textual "diffs."

Traditionally, the source code of software systems is compared using textual difference utilities. In the original `diff` utility (see Hunt and MacIlroy [16]), these comparisons are line-based, where each line gets hashed, which is then used in the calculation of a longest common subsequence. A second popular algorithm for string comparison is the Levenshtein distance [20]. Here, the minimum number of insertions, deletions, and replacements of characters is calculated to transform one string into another.

In the context of bioinformatics, similar approaches can be found in the works of Needleman and Wunsch [29] and Waterman, Smith, and Beyer [44].

Most of the above approaches share the idea of a longest common subsequence. Here, the common elements (in order, with insertions and deletions) to two sequences are calculated. To calculate a longest common subsequence, a sum of individual similarities or differences, as well as scores or penalties for insertions and deletions is optimised to a minimum or maximum. Waterman, Smith, and Beyer [44] use a more complicated scoring scheme, which allows the creation of arbitrarily sized gaps for a single insertion or deletion penalty. For the Levenshtein distance, for example, one needs the minimum sum of deletions and insertions of value 1, differences of value 1, and identities of value 0.

The traditional algorithm for these problems uses dynamic programming and calculates a matrix of values in $\mathcal{O}(mn)$ operations (m and n are the lengths of the sequences). For some situations, it is sufficient to store only parts of the matrix and to only provide the numeric solution (optimisation goal) of the algorithm. But in general, the whole matrix is stored to allow a calculation of a backtrace through the matrix and to provide the sequence of insertions, deletions, replacements, and matches.

The Needleman-Wunsch algorithm traverses the table line by line (or column by column), and for each element chooses the maximum score that can be achieved by combinations with the above, left, or above-left elements. Steps from positions above or left are insertions or deletions. Steps from above-left are a match or a mismatch, respectively, depending on the elements in question. Optionally, the chosen steps can be stored as well, allowing an easy calculation of the path taken and therefore of the resulting alignment.

Table 3.3: Needleman-Wunsch Table of the Alignment between 1,2,3,4,5,6 and 2,3,8,5,6

	–	1	2	3	4	5	6
–	**0.0**	**0.0**	0.0	0.0	0.0	0.0	0.0
2	0.0	0.1	**1.0**	1.0	1.0	1.0	1.0
3	0.0	0.1	1.0	**2.0**	2.0	2.0	2.0
8	0.0	0.1	1.0	2.0	**2.1**	2.1	2.1
5	0.0	0.1	1.0	2.0	2.1	**3.1**	3.1
6	0.0	0.1	1.0	2.0	2.1	3.1	**4.1**

As an example, we take two sequences of numbers, namely $1, 2, 3, 4, 5, 6$ and $2, 3, 8, 5, 6$. As scoring, we use insertions and deletions with a value of 0, a value of 1 for the match of two elements, and 0.1 for a mismatch. Table 3.3 shows the resulting score table, with an annotated backtrace through the matrix that gives the alignment. Such a table is constructed row-by-row. The first element in the first row shows the start score of 0.0. The row is then completed by aggregating the scores of each element of the first sequence as deletion (all 0.0). In the second row, the calculation is more complex. Here, the first element represents insertion of the first element of the second sequence (again, 0.0). Calculating the second element is more complex. It can be one of three (four, if one counts match and mismatch separately) choices: A match (1) or mismatch (0.1) between the elements 1 and 2 (here, mismatch) with a total score of 1 or 0.1, respectively, derived from the top-left prefix (0.0) of the position. Or, an deletion or insertion (both 0.1), either of 1 or 2, derived from the top or left elements (total 0.1), respectively. The algorithm chooses the case with the highest total score. Here, it is a mismatch. The third element in the second row follows the same scheme for the elements 2 and 2. As the elements are identical, the highest value can be reached with a match (1.0) derived from the top left (0.0). The other choices of adding 0.0 to the top (0.0) or left (0.1) cannot reach this score. The table is completed by repeating this scheme. By saving the choice for each table element (not shown), the alignment can be extracted by backtracking through the table (bold numbers).

An important observation about this algorithm is that, when the actual sequence is not needed, it is sufficient to only store the current and last row of the matrix to calculate the matrix and return the numeric optimisation result, which allows an implementation in linear space.

A variant to the classical algorithm is Hischberg's algorithm that solves the problem in the same time complexity, but only needs $\mathcal{O}(\min(m, n))$ in space complexity. This variant is important for the differential analysis of the ordered control-flow graph (OCFG), since executables can easily contain ten-thousands of functions. For example, the linux kernel contains about 30000 functions for the x86 architecture and a Needleman-Wunsch matrix between two kernel versions might not fit into main memory.

The idea (following the classical divide-and-conquer) is that it is possible to split the sequence alignment problem into two subproblems by splitting both sequences at a pair of elements from the resulting alignment. When the subproblems are solved, the original alignment can simply be obtained by concatenating the subproblem alignments.

It is possible to find a position to split at using the linear space variant of the Needleman-Wunsch algorithm, by choosing an arbitrary split in one sequence and then calculating an ideal split point of the second sequence by clever aligning the resulting subsequences of the first split with the second sequence. We omit the lemma and proof given by Hirschberg [15] as the precise workings of the algorithm are not important for this thesis. Nonetheless, we give an intuition about the split in the next example.

Figure 3.1 shows the recursion tree of an execution of Hirschberg's algorithm on the above example sequences. Each node shows both sequences. The first sequence is split at an arbitrary position, marked by the vertical line. A corresponding split is then calculated for the second sequence, and the subalignments are recursively calculated. The recursion terminates when only single elements are left in the sequences.

Figure 3.2 shows the calculation of the first split in the first step of the execution of Hirschberg's on the above example sequences. The input sequence is split into two sub-sequences and both sub-sequences are aligned to the full second sequence. For the first subsequence, this is done without any special change, for the second subsequence, both sequences are reversed. The calculated alignments (calls to Needleman-Wunsch) only return the last row of the alignment result (last row of Needleman-Wunsch matrix).

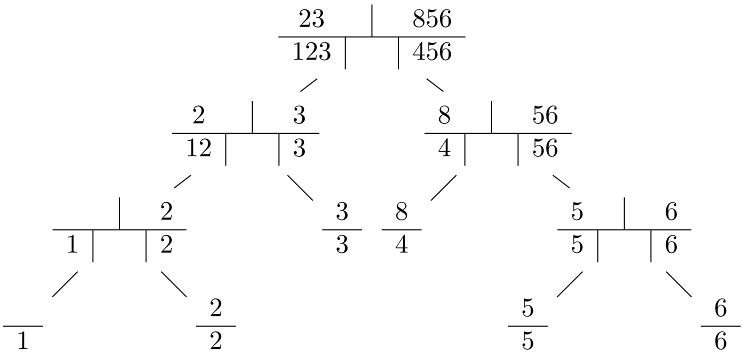


Figure 3.1: Example Recursion Tree of Hirschberg's Algorithm

```
// NW: Needleman Wunsch
scoreL =NW ([2, 3],[1, 2, 3, 4, 5, 6]);
// scoreL =[0, 0.1, 1, 2, 2, 2, 2];
scoreR =NW (rev ([8, 5, 6]), rev ([1, 2, 3, 4, 5, 6]));
// scoreR =[0, 1, 2, 2.1, 2.1, 2.1, 2.1];
sum =elem_add (scoreL, rev (scoreR));
// sum =[2.1, 2.2, 3.1, 4.1, 4, 3, 2];
take = arg_max(sum)
// take = 3
```

Figure 3.2: Example of Hirschberg's Split Calculation

A naïve interpretation of the resulting scores is possible. We first take a look at the scores of the subproblems, which both possess an increase-then-plateau effect, and then combine these plateaus to a single maximum that is the ideal split point. For the first alignment, the obviously best alignment is by pairing 2 and 3 with their counterparts. The result shows this, the first two elements of `scoreL` corresponding with alignments with no element and 1 respectively are low (0 (insertion) and 0.1 (mismatch)), the third and fourth element correspond to the matches of 2 and 3, and the score correspondingly sums up to 2. The last three elements are insertions or deletions again and do not change the score. Overall, an increase-then-plateau effect can be seen for our scoring scheme, where all the elements are ideally paired when the plateau is reached. Similarly, the second alignment calculates such a plateau, but starting from the other sides of the sequences. With these two plateaus, the ideal split point between the sequences can be determined by finding the position of the maximum (`take`) of an element-wise addition (`sum`) of the two results of the subproblems (with the second result reversed).

Hirschberg's algorithm for sequence alignment, introduced in this chapter, is used for our differential analysis in Chapter 6.

3.4.2 Executables

For executables, most differential analyses are concerned with identifying known parts of the executable, or identifying known (malicious) behaviour of an executable.

The basic idea is identifying known functions from a library. The most prominent software in this field is the fast library identification and recognition technology (FLIRT)[9] integrated in IDA Pro. This technique extracts masked byte-valued patterns from known libraries and applies these patterns to functions in an unknown executable. Masking of the byte sequence is necessary to exclude parts of the function where the linker inserts executable-specific addresses.

[9] Ilfak Guilfanov. *IDA Fast Library Identification and Recognition Technology (FLIRT Technology): In-Depth.* 2012. URL: `http://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml` (visited on 05/19/2017).

Another technique, by Stojanovic, Radivojevic, and Cvetanovic [40], uses information retrieval techniques to identify functions by querying a database of feature vectors containing several metrics of known functions.

An approach on an abstraction level between the two abovementioned is called BinDiff[10], from Dullien and Rolles [10]. Here, among others, comparisons of the control-flow graph structure are employed to identify the subset of similar functions between two sets of functions. As a side note, similar to our approach of calculating an alignment of two function sequences, BinDiff assumes subsequences of functions as identical, if a subsequence of identically sized function pairs is found that starts with an already identified function.

Another idea, from Han, Lim, Kang, and Im [14], uses hashing on the program opcodes to produce representative image matrices of a program, which are visually comparable between different executables. Similar to our approach for the graphical representation of the OCFG, a representative matrix is produced, but different to our approach, the position of each indicator in the matrix is not directly dependent on values in the program, but depends on a hash value to determine its position.

As abstraction over the concrete control-flow of executables, work exists that extracts application programming interface (API) call sequences from the executables (both statically and from execution traces). With these sequences, it is for example possible to detect suspicious behaviour of the software [36, 42], or even to analyse the behaviour in more detail [1]. Another approach from Cho, Kim, Shim, Park, Choi, and Im [7] uses the sequences to create similarity comparisons between different malwares.

[10] zynamics. *BinDiff*. URL: `https://www.zynamics.com/bindiff.html` (visited on 05/19/2017).

4 Ordered Control-Flow Graph

In this chapter, we introduce the ordered control-flow graph (OCFG), formalise it, and introduce its graphical representation.

4.1 Requirements

As stated in the introduction, this thesis introduces a structural analysis applicable to executables. To achieve a general applicability, *the representation must rely on low-level control-flow only*. This means, that the representation needs to be built on the association of instructions with addresses and the modelling of control flow through the use of a program counter that, after the execution of each instruction, contains the address of the next instruction (see Section 2.2). A common simplification of this data is the aggregation of linear sequences of instructions into basic blocks as discussed in Section 2.1.

In Section 2.4, we discussed which artefacts from the source code structure are preserved in the function sequence and are not affected through the order-preserving modifications. To exploit this information in the structural analysis, the *ordered control-flow graph (OCFG) must preserve this sequence of functions.*

In Section 3.2, we discussed the roles of patterns to allow humans to comprehend software whose complexity would otherwise exceed the limits imposed by the human brain [27]. To allow a human to recognise patterns, if they are present, *the OCFG must permit a graphical view.*
Additionally, for the later differential analyses, *the view must provide a variant to allow for comparisons of executable pairs.* For this it is important that similar executables with order-preserving changes result in recognisably similar views. In a combined view of the executables, it is necessary that both executables use compatible values to allow presentation on common scales.

```
bb = (start, len, {next})
call = (caller, callee)
f_c = (name, start, end, size, {bb}, {call})
data_c = {f_c}
```

Figure 4.1: Control-Flow Data from Disassembler

4.2 Formal Description

This section formalises the data used in the ordered control-flow graph (OCFG), and the processing involved, starting from control-flow data exported from a disassembler which is converted to an OCFG (a set of combined source and target tuples, called control accesses, with orders for source and target addresses).

In each of these steps, we pay special attention to preserving the sequence of functions in the executable. This order of functions is the main idea of the OCFG representation compared with a control-flow graph (CFG).

For the evaluations in this thesis, we extract the control-flow data from the IDA Pro disassembler. It would also possible to use the control-flow data from other sources like sweep disassemblers or even traced runs of a program, with minor changes.

Since IDA Pro uses a recursive descent disassembling strategy assisted by heuristics, this data is an unsound approximation of the real control flow, but has shown to be sufficient for the analysis so far. Note that, for some executables, the analysis was performed interactively in IDA Pro, to improve the recognition of function boundaries.

For each function, each basic block is identified and all of its known next blocks are extracted. Additionally, all function calls are extracted. The control flow from every single instruction to the next instruction is not extracted. While it is possible to also use this information, the effect for our later defined graphical view is negligible. Function starts and ends are extracted as they result from the IDA Pro analysis.

Figure 4.1 shows a formal notation of the extracted control-flow data in sets and tuples. The whole `data_c` control-flow data is a set of each function's `f_c` control-flow data. For each function, the function name (when available, not used in analyses, see next paragraph), the start and

```
f_n = (name, filename)
exe_n = { f_n }
```

Figure 4.2: Function to Source Code Relation (Ground Truth)

end addresses, and the function size in bytes are extracted. Additionally, a list of all basic blocks `bb`, as well as a list of tuples of all `caller`s and `callee`s are extracted. For each basic block, the start address, length in bytes, and the list of possible next basic blocks are extracted.

Additionally, the relation between function names and source files is extracted from the binaries using GNU's not unix (GNU) objdump. In the analysis, this name and source-code relation information is used neither in the construction of the OCFG and structural analysis nor in the differential analysis. However, it is used as ground truth of the evaluations of both analyses. Figure 4.2 shows the formal notation of the name and filename information in sets and tuples. For an executable `exe_n` a list of all functions is extracted. For each function, the function name `name` and, if available, the source file name `filename` are extracted.

We continue with natural language definitions of control accesses and the ordered control-flow graph.

Definition 1 (Control Access) *A control access is a pair of source and target addresses.*

A control access is either naturally associated with a jump or call in the program's control flow or it is a control accesses for a function entry point with identical source and target addresses, which are not necessarily included in the program's control-flow graph.

The source address of a control access can be presented as direct address, reduced to its (originating) function start address, or reduced to the (originating) function index. The target address of a control access can be represented as direct address as well or converted to an enumeration of the target addresses. Additionally, the enumerated targets can be scaled to be of zero-mean and unit-variance. Note that each of these transformations do not change the order of elements on their respective scale (source or target).

```
r_src = (src, fstart, fnum)
r_tgt = (tgt, tgtidx, stgt)
r_opt = (typ, name, filename, exe)
control_access = r_src . r_tgt . r_opt
OCFG = {control_access}
```

Figure 4.3: Ordered Control-Flow Graph: A Set of Control Accesses

Definition 2 (Ordered Control-Flow Graph) *The OCFG is a set of control accesses. The control accesses are independently ordered by their source and target addresses.*

The source and target parts of the control accesses are given in different representations. These representations are introduced here, and their usage in the graphic views is discussed in Section 4.4. Each control access also contains optional information not used in the analyses, but for the inspection of truth in the evaluation.

Figure 4.3 shows the OCFG representation in set and tuple notation. The data for the executable contains a set of control accesses. For readability, the control access tuples are split into three separate tuples that are concatenated. These contain the source, target, and source-code relation information, respectively.
The first component `r_src` contains the source information of a control access. `fstart` is the first address of the function, `fnum` is the index of the function in the executable, and `src` is the source address of the control access. The second component `r_tgt` contains the target information of a control access. `tgt` is the target address of the function's control access, `tgtidx` is the index of the control accesses target in the list of all control access targets, `stgt` is the `tgtidx` scaled to be of zero-mean and of unit variance. The last component `r_opt` contains additional information of a control access. `typ` is the type (entry, jump, or call) of the control access, `name` can contain the name of the function that the access belongs to, `filename` can contain the source filename associated with the function the control access belongs to, and finally `exe` contains the name of the executable that contained the control access. This information can be used for annotations in the views and is used to inspect the ground truth in the evaluation.

```
d = []
for f in join("name", data_c, exe_n):
        jumps = [(typ="jump", src=bb.start, tgt=bb.next) for
↪  i in f.bb]
        calls = [(typ="call", src=call.caller,
↪  tgt=call.callee) for i in f.call]
        entry = [(typ="entry", src=f.start, tgt=f.start)]
        # c is list of accesses per function
        c = concat(jumps,calls,entry)
        # set for all elements of function (whole column)
        c[,.fstart] = f.fstart; c[,.fnum] = f.fnum
        c[,.name] = f.name; c[,.filename] = f.filename
        d = concat(d,c)
# transform targets by column
d[,.tgtidx] = enumerate_unique(d[,.tgt])
d[,.stgt] = scale(d[,.tgtidx])
# set for all elements of executable
d[,.exe] = exe
```

Figure 4.4: Pseudocode of Conversion from Disassembler Data to the Ordered Control-Flow Graph

```
void A(void); void B(void);

int nondet(void) {
  int a; return a;
}

void _start(void) {
  if( nondet() ) A();
  B();
  asm("mov $60, %rax");
  asm("mov $0, %rdi");
  asm("syscall");
}
```

(a) main.c

```
void A(void) { return; }
```

(b) a.c

```
void B(void) { return; }
```

(c) b.c

```
LDFLAGS=-nostartfiles
LDFLAGS+=-static
CFLAGS=-O0
main: main.c b.c a.c
```

(d) Makefile

Figure 4.5: Input Source Files and Makefile

From control-flow data `data_c` and names for the executable `exe_n`, we calculate the representation as a simplified flat version, which we formally specify in the next paragraph. Figure 4.4 shows the pseudocode of the conversion algorithm. All data for an executable is aggregated in `d`. Control-flow data and source-code relation information is joined on the common name field. We iterate over this data (per function) and collect the intraprocedural jumps between basic blocks, all calls to other functions, and the function entry point. Additionally, we annotate the function start address, number, name, and the filename from the source code relation information. Finally, we save the name of the executable and set the values of the `tgtidx` and `stgt` fields.

4.3 Fully Worked Example

To aid comprehension, we provide a complete example, from source code to the ordered control-flow graph (OCFG). We already show the example in the final view as proposed for the OCFG although the views are still to be defined (see next section).

Table 4.1: Ordered Control-Flow Graph of `main`

r_src		r_tgt			r_opt			
src	fnum	tgt	tgtidx	stgt	typ	name	f'name	exe
0x40010c	1	0x40010c	1	-1.4	entry	nondet	main.c	main
0x400115	2	0x400115	2	-0.9	entry	_start	main.c	main
0x400119	2	0x40010c	1	-1.4	call	_start	main.c	main
0x400120	2	0x400122	3	-0.4	jump	_start	main.c	main
0x400120	2	0x400127	4	0.2	jump	_start	main.c	main
0x400122	2	0x400127	4	0.2	jump	_start	main.c	main
0x400122	2	0x400146	6	1.2	call	_start	main.c	main
0x400127	2	0x40013f	5	0.7	call	_start	main.c	main
0x40013f	3	0x40013f	5	0.7	entry	B	b.c	main
0x400146	4	0x400146	6	1.2	entry	A	a.c	main

Figure 4.5 shows the source code used for a mini example of a standalone static linux executable. The files `a.c` and `b.c` contain one function each, which are called `A` and `B`, respectively. Both functions do nothing and just return immediately. The file `main.c` defines two functions, the first (`nondet`) just returns a number, the second (`_start`) calls this function and uses this number to decide whether to take the branch of the `if` and call `A` first and then `B` or only call `B`. It then performs a `syscall` to perform `sys_exit` using assembly statements. The provided Makefile contains data for an implicit rule that compiles the files `main.c`, `b.c`, and `a.c` into an executable called `main`. It uses the linker flags `-nostartupfiles` and `-static` to create this executable. These allow the creation of a minimal executable where `_start` is directly executed.

Figure 4.6 shows the dump of the `.text` section of the created executable. It only contains the four functions from the source files. Since neither optimisation was used during the compilation, nor dead-code elimination or other simplification are performed, the disassembled code basically matches the C source code.

By walking through the control-flow graph (see Figure 4.7) of the disassembled code, the data for the plots can be extracted. Table 4.1 shows the data using the OCFG tuple representation. The columns follow the

```
main:     file format elf64-x86-64


Disassembly of section .text:

000000000040010c <nondet>:
  40010c:        55                        push   %rbp
  40010d:        48 89 e5                  mov    %rsp,%rbp
  400110:        8b 45 fc                  mov    -0x4(%rbp),%eax
  400113:        5d                        pop    %rbp
  400114:        c3                        retq

0000000000400115 <_start>:
  400115:        55                        push   %rbp
  400116:        48 89 e5                  mov    %rsp,%rbp
  400119:        e8 ee ff ff ff            callq  40010c <nondet>
  40011e:        85 c0                     test   %eax,%eax
  400120:        74 05                     je     400127 <_start+0x12>
  400122:        e8 1f 00 00 00            callq  400146 <A>
  400127:        e8 13 00 00 00            callq  40013f <B>
  40012c:        48 c7 c0 3c 00 00 00      mov    $0x3c,%rax
  400133:        48 c7 c7 00 00 00 00      mov    $0x0,%rdi
  40013a:        0f 05                     syscall
  40013c:        90                        nop
  40013d:        5d                        pop    %rbp
  40013e:        c3                        retq

000000000040013f <B>:
  40013f:        55                        push   %rbp
  400140:        48 89 e5                  mov    %rsp,%rbp
  400143:        90                        nop
  400144:        5d                        pop    %rbp
  400145:        c3                        retq

0000000000400146 <A>:
  400146:        55                        push   %rbp
  400147:        48 89 e5                  mov    %rsp,%rbp
  40014a:        90                        nop
  40014b:        5d                        pop    %rbp
  40014c:        c3                        retq
```

Figure 4.6: Disassembly of `main` Executable

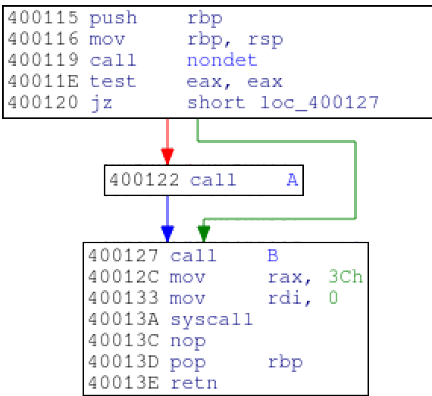


Figure 4.7: Control-Flow Graph of `_start` Function

definition of the internal representation in Definition 2. Each function is represented with at least an entry of type entry which marks the entry point of the function. Additional entries are of type jump or call for jumps to other basic blocks or calls to other functions, respectively.

To provide a feeling for the address data in Table 4.1, Figure 4.8a shows a plot of the `src` and `tgt` columns. The background of the plot shows the partitioning of the `src` address space among the functions. Both axes use a normal numerical scale for the addresses.
Each dot in the plot corresponds to one row of the table. For the functions `nondet`, `A`, and `B`, this is just one dot each, for each function's single basic block. For the `_start` function, there are several dots. Most of the dots represent the control flow between the basic blocks of the function. The dot to the very left and the two dots to the very right in the area of the `_start` function represent the calls to the `nondet`, `A`, and `B` functions.

Figure 4.8b uses the source function index (`fnum`) for functions (vertical) and target address index `tgtidx` for accesses (horizontal). This is the normal presentation of the data, that will be defined in the next section as View 1. It provides improvements for the structural analysis (to be discussed there) over the simple address scales for executables of realistic sizes.

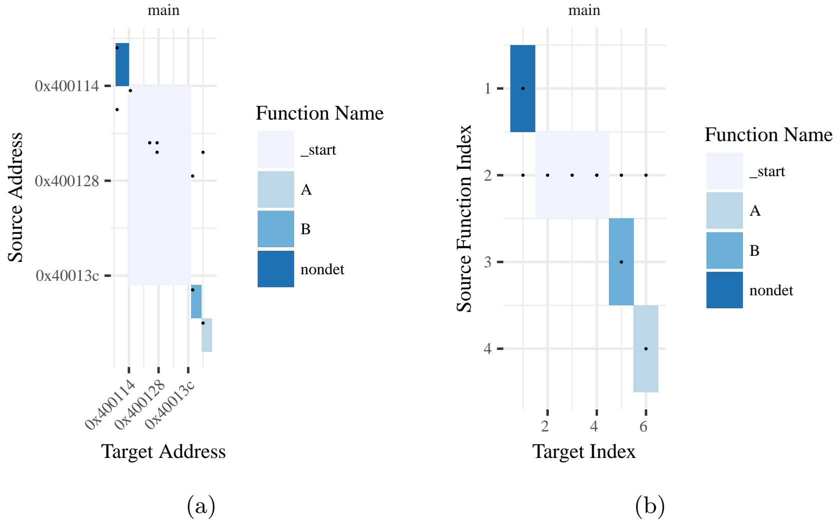


Figure 4.8: Graphical View of the Ordered Control-Flow Graph of `main`

4.4 Graphical View of Ordered Control-Flow Graphs

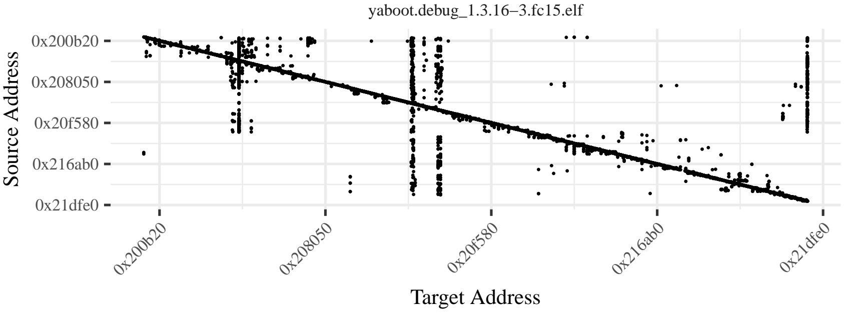
The graphical representation of the ordered control-flow graph (OCFG) uses pairs of source and target addresses in a scatter plot for each control access. We use the vertical axis for source information and the horizontal address for target information. In the final proposed views, each row is associated with a function.

Section 4.2 prepared the data to contain the source function addresses and source function indices, as well as the target addresses, target address indices, and scaled target address indices. In this section, we explore the suitability of unprocessed addresses (`src`, `fstart`, and `tgt`) for a graphical presentation and explain the improvements the preprocessed representations (`fnum`, `tgtidx`, and `stgt`) imply for the view. This section is organised into three parts, the first that discusses representations of control access sources, the second that defines a view for single executables, and the last that defines a view for executable pairs. We use `yaboot`[1] and `doom`[2] as example executables.

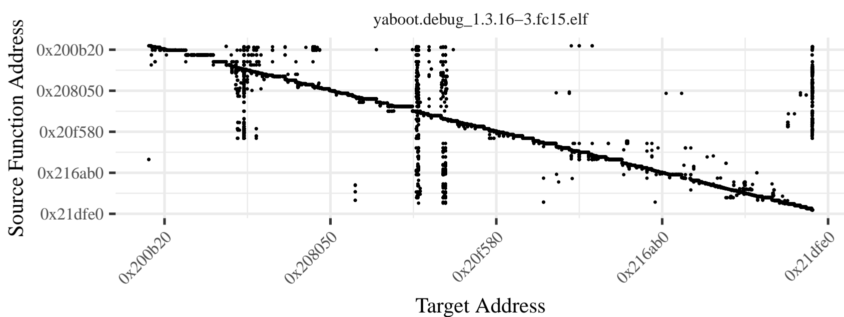
We first have a look at an intermediate view to explain the representation of the control access sources. Figure 4.9 shows three plots of the same executable. Both use unprocessed target addresses `tgt`. For the source functions, Figure 4.9a uses unprocessed addresses `src`, Figure 4.9b uses function start addresses `fstart`, and Figure 4.9c uses function indices `fnum`.

In the completely unprocessed view, the program addresses are directly visible. Here, the intraprocedural control flow forms a diagonal through the plot (see Figure 4.9a).
By using the function start addresses, parts of the diagonal belonging to larger functions are separated into horizontal lines (see Figure 4.9b).
The use of function indices hides the program's address space and closes the gaps introduced in the address steps between the function start addresses. This reshapes the diagonal across the plot to represent the function lengths

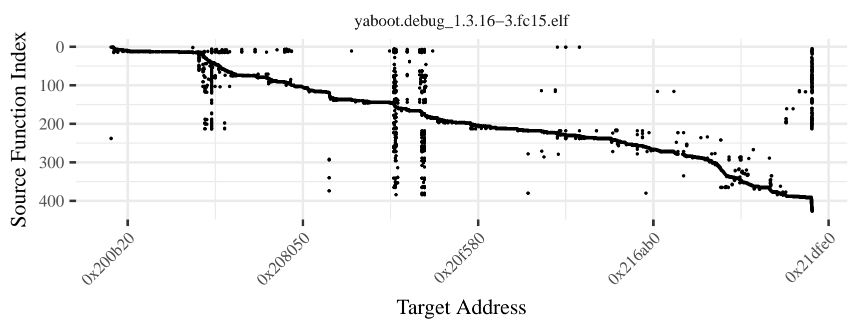
[1] *Yaboot*. URL: `https://web.archive.org/web/20160312033912/http://yaboot.ozlabs.org/` (visited on 05/19/2017).
[2] *Doom*. URL: `https://en.wikipedia.org/wiki/Doom_(1993_video_game)` (visited on 05/19/2017).



(a) Source Addresses



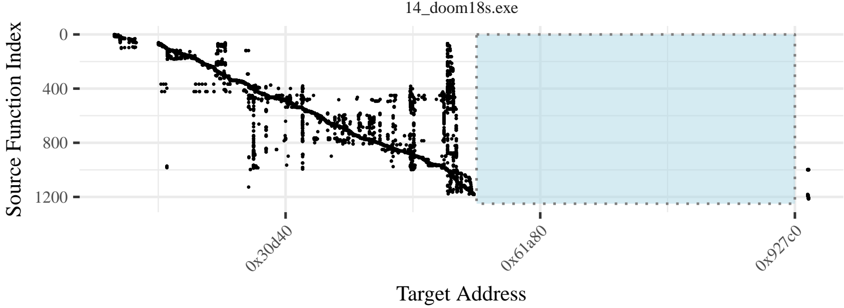
(b) Source Function Addresses



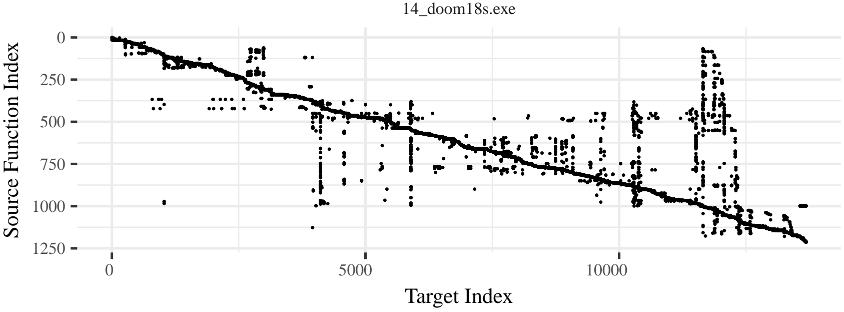
(c) Source Function Address Indices

Figure 4.9: Intermediate: Source Function Data Preparation

(the number of bytes occupied by the function's instructions). This allows large functions as well as sections of small functions to immediately emerge from the plot. For example, see the flat part of the diagonal for function number 10 in Figure 4.9c. For the final view we use the source function indices because of their deformation effects on the diagonal.



(a) Target Addresses



(b) View 1: Target Indices

Figure 4.10: View for Single Executables

Now, we develop the first view for single executables. Figure 4.10 shows two plots of the same executable. Both use source function indices `fnum`.

For the target addresses, Figure 4.10a uses unprocessed addresses `tgt`, while Figure 4.10b uses indices `tgtidx`.

In most cases, the two representations look very similar, because the target addresses are usually packed well, but in some cases, there are gaps between executable parts in the address space. Figure 4.10a shows such a gap in the second half of the target address space up to `0x927c0`. It forces most of the representation to be squeezed into a fraction of the available space. Figure 4.10b shows the usage of indices to close such gaps.

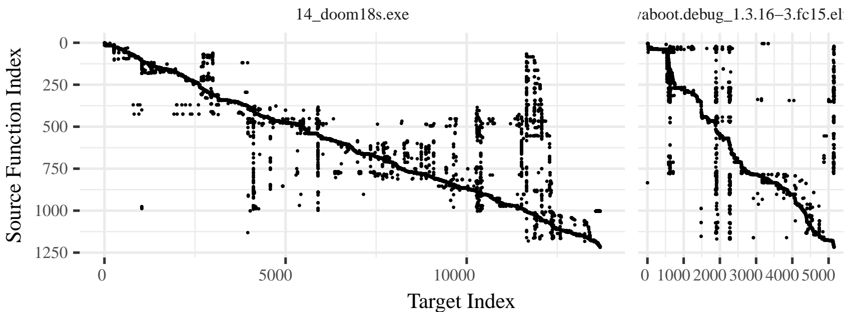
View 1 (Single Executable or Single OCFG)
Scatter plot of source function indices `fnum` against target indices `tgtidx`.

Figure 4.11 shows two plots of aligned executable pairs. Both plots use source function indices `fnum` and show the same two executables. For the target addresses, Figure 4.11a uses target indices `tgtidx`, Figure 4.11b uses scaled targets `stgt`. The plot shows an alignment of the executables calculated by the later presented differential analysis (DA). This allows the presentation of the executables on the same source function index scale, although one executable has about 400 and the other about 1200 functions.

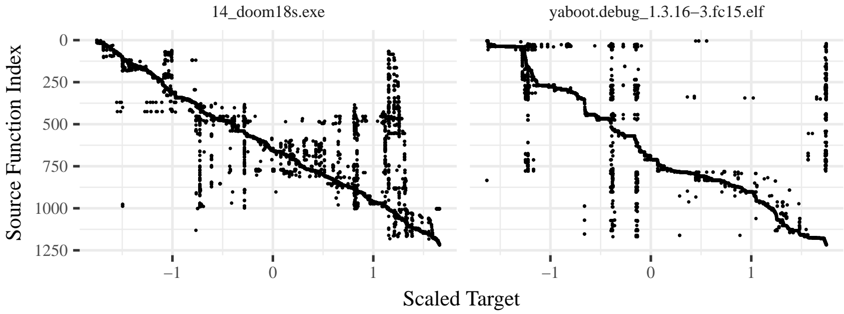
Figure 4.11a shows that between different executables the density and number of accesses for a given target address can be very different. In this case, the accesses of the right executables are disproportionally squeezed into a very narrow area, making a comparison of the patterns of executables difficult.
Figure 4.11b uses target address indices scaled to be zero-mean and of unit variance. This hides the density of access targets and allows a comparison of two executables on the same target address scale. The presentation in this figure uses two adjacent plots. Using scaled target address indices, it is possible to overlay two OCFGs in a single plot.

View 2 (Executable Pairs or OCFG Pairs)
Scatter plot of function indices `fnum` against scaled targets `stgt`.

In summary, we recommend using source function indices for all plots, and target address indices for plots of single executables or OCFGs (View 1) and scaled target address indices for plots of executable or OCFG pairs (View 2).



(a) Target Indices



(b) View 2: Scaled Targets

Figure 4.11: View for Aligned Executable Pairs

5 Structural Analysis for Program Comprehension

In this chapter, we describe how different statement-level accesses are represented in the ordered control-flow graph (OCFG), introduce 9 visual patterns, inspect and test the patterns using a set of example executables, and apply the patterns in a case study. With the control-flow data in the OCFG and its proposed views, it is possible to leave the statement-level abstraction of individual jump or call accesses, and abstract each function to a set of access targets. This allows the analysis of control dependences between pairs of functions or sets of functions.

In the next paragraphs, we discuss the representations of interprocedural and interprocedural control-flow in the graphical view, and in the context of program comprehension of the executable.

In the OCFG representation, we order the functions by the first address of their appearance in the executable. This choice allows exploiting the locality of the code, preserved from the source code's structure, to present visual structures that correspond to function and module roles and partitions, which can be recognised and interpreted by an analyst. In Sections 3.1 and 3.2, we discussed the artefacts commonly used for program comprehension (PC), and how the lack of higher-level artefacts challenges the human cognitive limit. In the following, we introduce several patterns that highlight a program's control flow to aid structural analysis of an executable, without challenging the cognitive limit. In Sections 2.2 and 2.4, we discussed that the sequence of functions within an executable is determined by the order of modules in the build process and stays stable under several order-preserving modifications. We use this property to define patterns on function sets that allow the identification of program modules, and coupling between the modules.

To provide an intuitive analogy to the graphical view of the OCFG and the patterns to be introduced, we look at cross references in a textbook. Here, many paragraphs contain references to other parts of the work, and a natural order of the paragraphs, their order of occurrence in the book, exists. Also, each paragraph spans a number of text lines in the book. We can also assume that we can assign a unique number to each line of text in

the order of occurrence. So, analogously to the OCFG, we collect coupling data in paragraph vs. line number format: Intra-paragraph data for the lines of each paragraph and cross reference (inter-paragraph) data from each paragraph to the lines of all other paragraphs. In a scatter plot where this data is plotted per paragraph against the line numbers, each line consists of a horizontal part that represents the lines of text for one paragraph, and additional dots that represent the cross references of the paragraph.
In this plot, clusters and patterns emerge in the references, as there exists a locality in the paragraphs, and the references will only target limited parts of the rest of the book. For example, in a mathematics textbook, the paragraphs reference the names of variables in their corresponding formulas locally, and may depend on some definitions given in another part of the book.

In terms of the executables, paragraphs correspond to the functions, the references to function calls, and the clusters and patterns to the coupling between code modules.

Next, we discuss several control properties on function and function set level. In the following, we use a notion of address range, which also includes ranges of function numbers or start addresses, since these also just represent address ranges.

Functions The major part of control accesses of an executable on statement level is made from intraprocedural control flow, namely standard structured programming constructs like `if`, `for`, or other control-flow constructs. Still, the OCFG's views abstracts the intraprocedural control flow to one horizontal line segment proportional to the function size for each function. Note that, in the detailed analysis of executables, the intraprocedural control flow plays a major role. For example, reading and understanding of a function's control-flow graph by an analyst is necessary to comprehend a function. But, summary metrics of the control flow, like the McCabe or Halstead complexity, are closely correlated to the function size. So, even if our view hides most of the intraprocedural data, some information is still present.

In the executable, the intraprocedural data covers all of the addresses occupied by code. This results in the complete coverage of the source and target addresses by intraprocedural data in the OCFG. Since the source

accesses for all functions are shown in order of their appearance in the executable, and the target addresses share the same order, albeit on a different scale, this intraprocedural data forms the diagonal in the plot. By grouping of source addresses to function start addresses or functions numbers, a deformation is introduced to the diagonal, which is proportional to the number of the addresses occupied by a source function or function size.

The second part of control accesses in an executable on statement level is interprocedural control flow, namely calling other functions and returning from these. In the OCFG, these show up at various positions in the address ranges. Here, special patterns in these appearances are of interest.

On the function level, some functions with high fan-in or fan-out usually play important roles in their executable. High fan-in means that control accesses for a function's start address cover large parts of the source address space. High fan-out means that a function has control accesses to many target addresses.

Function Sets In the OCFG representation, source files (or modules) are represented as sets of functions. If the stability assumptions are not broken, these sets of functions cover contiguous ranges in the address space.

A high coupling by calls between the functions is typical for functions local to a software module. Commonly, such coupling is defined as a simple relation and read in both directions. In the OCFG, the coupling is directional and follows the direction of the calls. It would be possible to make the interprocedural coupling information in the OCFG reflexive, but this is not desirable, as some information would be lost. In the OCFG such calls in a set of functions are represented by control accesses where the source and target address ranges cover the same range, but do not span the whole address range.
The directionality of the information allows an interpretation of the module construction in the graphical view. A module with an upper-right triangular coupling is constructed top-down in its source file, and the first function calls the latter, and so on. Similarly, a module with lower-left triangular coupling is constructed bottom-up.

When one module depends on another, there is a close coupling between two sets of functions. In the OCFG, such a dependency is represented by

Table 5.1: Interpretations of Visual Structures

	#	Structure	Interpretation
Fct.	1	Deformed Diagonal	Intra-Procedural Control Flow
	2	Global Horizontal	Fan-Out or Dispatch
	3	Global Vertical	Fan-In or Library
Module	4	Box or Triangle on Diagonal	Intramodular Control Flow
	5	Box or Triangle not on Diag.	Module Coupling
	6	Local Horizontal	Dispatch Function
	7	Local Vertical	Helper Function
	8	Vert. Space	No Incoming Inter-Mod. Coupl.
	9	Hor. Space	No Outgoing Inter-Mod. Coupl.

control accesses where the source addresses cover the first set of functions (outgoing), and the target addresses cover the second set of functions (incoming). Here, the directionality of the information allows the construction of dependency information, and ultimately the identification of top-level and bottom-level modules.

In addition to the properties of high fan-in and fan-out on the function-level (discussed above), it is possible that such a function's reference scope is local to a module. In the OCFG their fan-in or fan-out ranges are limited to their own or another module's address range.

Lack of coupling is a property that is useful for patterns on a function set level. The absence of coupling means that there are no control dependencies between one function and another. In the OCFG the absence is represented by non-existent control accesses to a given address range, except for the intramodular control accesses.

Table 5.2: Common Structure of Pattern Descriptions

Entry	Content
(Sketch)	A sketch that summarises the pattern's structure.
Name	The name of the pattern that summarises the pattern's shape.
Content	The software elements that the pattern highlights.
Level	The level of data presented. In this case, the level of control flow represented by the pattern.
Result	The data content of the pattern's interpretation.
Executables	Example instances of the pattern in executables.
Structure	A description of the pattern's structure and syntax of the patterns elements.
Interpretation	A description of the translation between the pattern's structure and syntax to the data content that the pattern represents.
Examples	Abstract examples of the pattern.

5.1 Pattern Definitions

The control coupling properties for source code and in the ordered control-flow graph (OCFG) as described in the last section each have a typical appearance in the view of the OCFG. This section gives an overview of the structures that appear in our plots of executables and discusses their interpretations. Table 5.1 gives an overview of the structures and interpretations for the different levels of structures contained in the view. There are function and module level patterns. Some pattern variants appear on both levels.

Each pattern is presented using a template of paragraphs for each pattern: We summarise each pattern using a record card style header and discuss the details in several sections (see Table 5.2). This style is inspired by the pattern definitions from Gamma [11]. Each of the descriptions contains links to the next section where the patterns are applied to a broad set of executables. These occurrences of the patterns serve as a test of the real-world application of the structural analysis.

5.1.1 Function Patterns

	1: *Deformed Diagonal*
Content	Intra-Procedural Control Flow
Level	Functions
Result	Locality of Function Sizes
Executables	1a and 2a

Structure Every plot contains a diagonal, which consists of the intraprocedural control flow. The deformed diagonal pattern is concerned with parts of the diagonal where the slope of the diagonal is high or low, resulting in steep or flat parts.

Interpretation The local slope of the diagonal implies the relative size of the functions. A flat part of the diagonal encodes a function of larger than average size, a steep part encodes a function of lower than average size.

Examples Examples of large (flat) functions include library functions like `printf`, and top-level application functions, which include parsers or decision logic to distribute work to different parts of an executable. Small (steep) functions can be helper functions from the library, like automations of function prologues, and epilogues. Other typical types of small functions are wrappers and getter or setter functions.

	2: *Global Horizontal*	
	Content	Fan-Out or Dispatch
	Level	Function
	Result	Global Fan-Out Hotspot
	Executables	4b and 2c

Structure The global horizontal pattern is concerned with horizontal line structures in the plot that span large parts of the reference (target) space up to most of the reference space and originate from a small number of functions.

Interpretation Horizontal structures that span large parts of the target address spaces represent functions that call a large amount of other functions.

Examples Examples of functions with large fan-out or fan-out over the whole program are top-level modules that call into the different parts of the program, shell or network interfaces, or parsers that dispatch tasks to other parts of the program.

	3: *Global Vertical*	
	Content	Fan-In or Library
	Level	Function or Module
	Result	Global Fan-In Hotspot
	Executables	4a, 3c, and 1c

Structure The global vertical pattern is concerned with vertical line structures in the plot that span large parts of the function space.

Interpretation Vertical structures that span large parts of the function space represent calls to functions from a large number of other functions spread over the whole program.

Examples Examples of functions with a large fan-in are the string and printing functions from the C-library. These functions are generally called by a high number of other functions from different places in the program. Functions from a library that provide the core of an application, such as an implementation of a custom arithmetic library, are also typically called from a high number of places in the program.

5.1.2 Module Patterns

	4: *Cluster on Diagonal*
Content	Intramodular Control Flow
Level	Modules
Result	Module Microstructures
Executables	3a to 6a

Structure Most plots contain clusters of references very near the diagonal that correspond to the intramodular control flow. This pattern is concerned with microstructures near the diagonal that usually appear as left-lower triangles, right-upper triangles, or boxes.

Interpretation Microstructures directly next to the diagonal provide information about the way the programmer has organised the functions in the source file. Since the order of functions in the source file is usually preserved, the plot can be used to inspect this organisation.

Examples For example, in a bottom-up design, the programmer accumulates leaf functions that get composed into higher-level functions. Such an organisation results in a left-lower triangle next to the diagonal. Following this rule, a top-down design where the source file starts with the final composition results in a right-upper triangle. Other organisations like sorting by alphabet then result in an unsorted module in the executable, which results in a box at the diagonal.

	5: *Clusters not on Diagonal*
Content	Intermodular Control Flow
Level	Modules
Result	Module Coupling
Executables	6a and 1b

Structure Most plots contain intermodular control-flow references that are not on the diagonal. This pattern is concerned with such clusters, which often take the shape of diagonals or boxes.

Interpretation Distinct clusters of references represent the coupling between two modules.

Examples For example, when one module wraps most of the functions of another module (and the wrappers have the same order as the wrappees), the plot contains a diagonal structure. When a module composes more complex functionality using another module, the plot contains a box-like structure, as each composition can reference multiple elements.

	6: *Local Horizontal*
Content	Dispatch Function
Level	Function
Result	Local Fan-Out Hotspot
Executables	5b and 4c

Structure The local horizontal pattern is concerned with horizontal line structures in the plot that span small parts of the reference space and originate from a small number of functions.

Interpretation Horizontal structures that cover parts of the target address spaces represent functions that call functions of a part of the program.

Examples An example of such a function is a dispatch function that distributes work within a single module.

	7: *Local Vertical*
Content	Helper Function
Level	Function
Result	Local Fan-In Hotspot
Executables	2b

Structure The local vertical pattern is concerned with vertical line structures in the plot that span one or more smaller parts of the function space.

Interpretation Vertical structures that span one or more smaller parts of the function space represent calls to functions from other parts of the program.

Examples An example of a function with large fan-in from a part of the programs is a helper function that encapsulates a functionality that is used from this part of the program, like an addition function in an arithmetic module.

	8: *Vertical Space*
Content	No Incoming Intermodular Coupling
Level	Modules
Result	Top-Level Module
Executables	6c

Structure Some plots contain areas where nearly no references are found. This pattern is concerned with vertical areas without references that span the whole program space above and below modular structures.

Interpretation Reference-free areas highlight modules without incoming intermodular coupling.

Examples For example, a top-level module of a program, although it might have a high cohesion (see Pattern 4), usually is not called from within the program itself, so there are no references to the code, resulting in space above and below the module.

	9: *Horizontal Space*	
	Content	No Outgoing Intermodular Coupling
	Level	Modules
	Result	Bottom-Level Module
	Executables	3b

Structure Some plots contain areas where nearly no references are found. This pattern is concerned with horizontal areas without references that span the whole target reference space around modular structures.

Interpretation Reference-free areas highlight modules without outgoing intermodular coupling.

Examples For example, a bottom-level utility module, like implementations of software floating-point arithmetic or other independent library functions, has no coupling to other parts of the software.

5.2 Pattern Examples

In this section, we inspect examples of the patterns in the executables, to show the general applicability of the patterns. To provide a wide range of executables we use `bzip2`, `gcc`, `povray`, and `perlbench` executables from CPU2006, `glibc`, and `vmlinux` (x86). The executables from CPU2006 represent different hosted executables as they can be found on most common machines. We include `glibc`, a C library, which is part of the runtime image of nearly all hosted software, as an example of a linked-in library. Lastly, we include the `vmlinux` kernel, which is a standalone executable that operates without a hosting environment, as an example of an embedded executable. Each of the pattern occurrences presented in the following subsections is manually checked against the source-code information contained in the executable's debug information.

5.2.1 Hosted Executables

We begin the set of pattern examples with hosted executables. These are the most prevalent executables on modern personal computers.

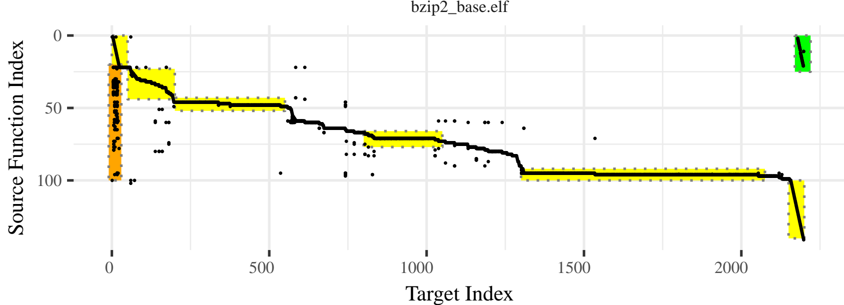


Figure 5.1: Patterns in `bzip2` Executable

Pattern Example 1: `bzip2` (see Figure 5.1)
The `bzip2` executable is very small, and has a small number of application level functions and a relatively large part of runtime and library glue.

Occurrence (1a): Deformed Diagonal (Pattern 1, Yellow)

Location	Interpretation
Functions 0 to 21, 100ff steep	Very short functions, such as wrappers
Functions 22 to 45	Short functions
Functions 46, 48, 71, 95, and 96 flat	Long functions, such as high-content functions

Inspecting the source-code relation information, the functions 0 to 21 and 100ff are function wrappers to call the dynamically linked C-library. 23 to 29 are short startup functions to instantiate the hosted process. 30 to 44 are functions from the SPEC benchmark instrumentation.
The names of the large functions are: 46: `fallbackSort`, 48: `mainSort`, 71: `BZ2_bzDecompress`, 95: `BZ2_compressBlock`, and 96: `BZ2_decompress`.

Occurrence (1b): Clusters not on Diagonal (Pattern 5, Green)

Location	Interpretation
Functions 0 to 25 and references 2170 to 2220	The cluster has a diagonal shape, the same as the associated functions (reference range 0 to 100, and function range 100ff, described in the last pattern occurrence), which shows a wrapper structure.

The functions from 0 to 21 each call a single function in the range of functions 100ff., the functions in both areas bear names from the C-library.

Occurrence (1c): Global Vertical (Pattern 3, Orange)

Location	Interpretation
Functions 30 to 100 and reference index 20	High fan-in functions, such as helpers or library.

Inspecting the function names from the source-code relation information, the references around 20 correspond to the functions 1 to 21, which are from wrappers to functions like `printf`, `free`, and `open`.

Pattern Example 2: `gcc` (see Figure 5.2)
The `gcc` executable is a large executable, which nonetheless shows visible changes on the diagonal, even on the small scale of the picture. On this scale, it is only possible to give rough references to the functions.

Occurrence (2a): Deformed Diagonal (Pattern 1, Yellow)

Location	Interpretation
Functions 2100 to 2200: flat	Large functions
Functions 2200 to 3150: steep	Small functions
Few functions around 3150: flat	Very large functions

Inspecting the source-code relation information, the functions between 2086 and 2174 seem to be concerned with the parameters of individual instructions for several processors, which are implemented using larger functions, the functions 2175 to 3121 with the generation and output of single instructions

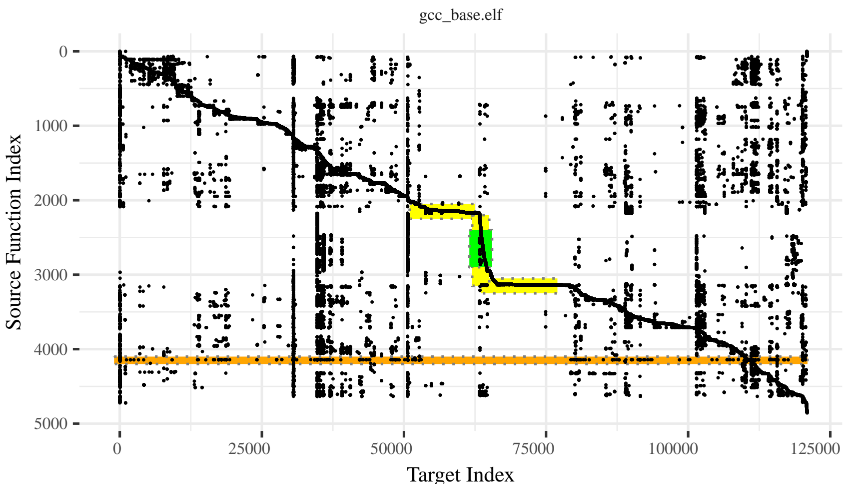


Figure 5.2: Patterns in `gcc` Executable

using smaller functions, and 3122 to 3157 with recognition, splitting, and copying of instructions using large functions.

Occurrence (2b): Local Vertical (Pattern 7, Green)

Location	Interpretation
Functions 2400 to 2900 and around reference 63000	Large intramodular fan-in

Inspecting the function names from the source-code relation information for these references, the functions begin with the string `gen_`. These are intramodular references from the whole set of instruction generation functions, which use smaller implementation parts.

The next pattern occurrence is hard to spot in the executable, but since top-level functions that actually call into most parts of the executable and to not use a hierarchy to distribute the work are seldom, it is included. Occurrence (2c): Global Horizontal (Pattern 2, Orange)

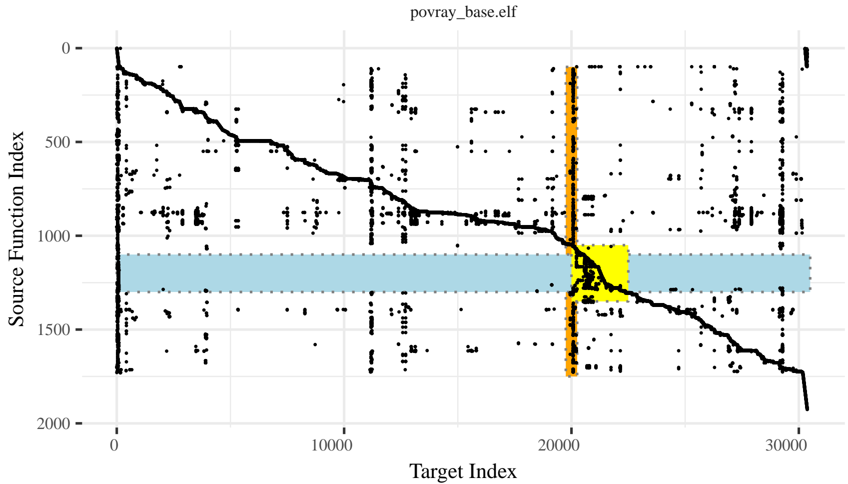


Figure 5.3: Patterns in `povray` Executable

Location	Interpretation
Function 4150	Large program level fan-out

Inspection of the source-code relation information shows the function 4143 to be `toplev_main`.

Pattern Example 3: `povray` (see Figure 5.3)
The `povray` executable is a medium sized executable. `povray` is a ray tracing renderer that reads a geometry, light, and camera description and approximates the camera's picture.

Occurrence (3a): Cluster on Diagonal (Pattern 4, Yellow)

Location	Interpretation
Function 1100 to 1300, left-lower triangle	Bottom-up module construction

Inspecting the names from the source-code relation information the functions 1051 to 1300 contain memory management, message queueing system, stream handling, other utility and file handling functions.

Occurrence (3b): Horizontal Space (Pattern 9, Blue)

Location	Interpretation
Function 1100 to 1300, limited number of references left and right	Bottom level module, not utilising other code or libraries

Like in Occurence 3a, inspecting the names from the source-code relation information, the functions 1051 to 1300 contain memory management, message queueing system, stream handling, other utility, and file handling functions.

Occurrence (3c): Global Vertical (Pattern 3, Orange)

Location	Interpretation
Reference index ca. 20100, spanning from functions 100 to 1750	Functions called from most parts of the program.

Inspecting the source-code relation information the reference indices 20071 and 20084 can be identified as `pov_malloc` and `pov_free`.

Pattern Example 4: `perlbench` (see Figure 5.4)
The `perlbench` executable is a medium sized executable. `perlbench` contains the general purpose Perl script language. Note that, although this is a suboptimal choice, all three occurrences use orange as indicator colour, to stay consistent with the colours used in the pattern definitions.

Occurrence (4a): Global Vertical (Pattern 3, Orange)

Location	Interpretation
References 29000 and 33500, called from functions 100 to 1875	Functions or modules used from most parts of the program.

Inspecting the source-code relation information the function names between references 29826 and 33017, corresponding with functions 1272 to 1413, contain the string `Perl_sv_`. The perl source tree reveals that the `sv.c` module implements the handling of scalar values.

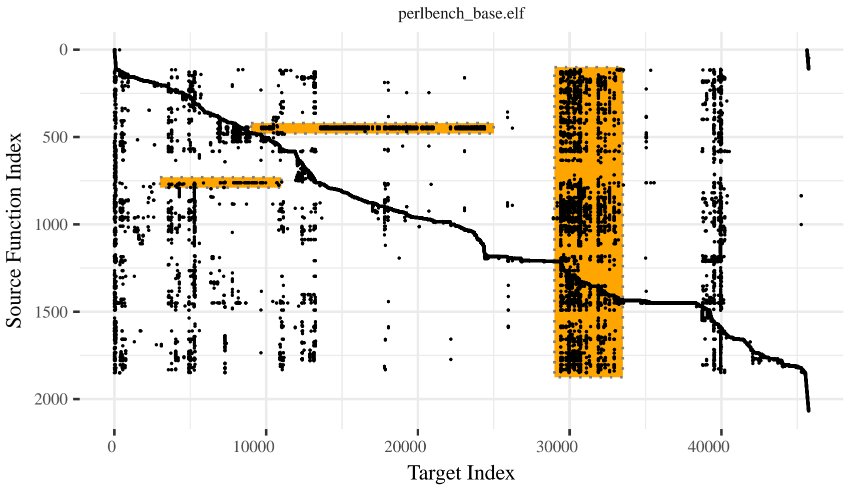


Figure 5.4: Patterns in `perlbench` Executable

Occurrence (4b): Global Horizontal (Pattern 2, Orange)

Location	Interpretation
Function around 450 targets functions ranging between indices 9000 to 25000	High fan-out dispatch function to several modules

By inspecting the source-code relation, the functions 443, 444, and 447 can be identified as `Perl_convert`, `Perl_newUNOP`, and `Perl_newBINOP`. These functions are part of the parsing functionality and reference a high number of operations or conversions depending on the input data, by calling the respective handlers.

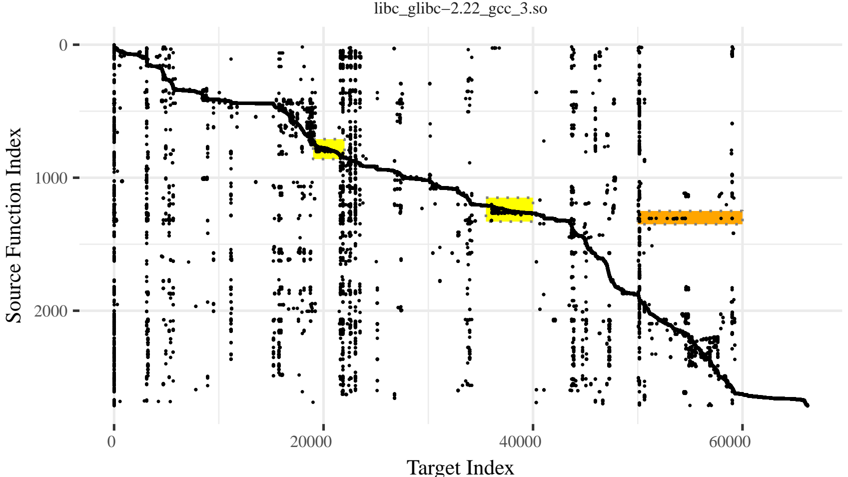


Figure 5.5: Patterns in `glibc` Executable

Occurrence (4c): Local Horizontal (Pattern 6, Orange)

Location	Interpretation
Function 760 accesses target address indices between 3000 and 11000	High fan-out dispatch function to a few modules

The source-code relation information names the function 762 `Perl_yyparse`, which refers to a group of handling functions that perform the necessary work. Incidently the functions around 450 for unary and binary operations, which we discussed earlier, are among these functions.

5.2.2 Runtime Library

Many executables depend on libraries. Such libraries can be statically linked in or added by a runtime linker. As an example of such a runtime library, we use `glibc`.

Pattern Example 5: `glibc` (see Figure 5.5)
The `glibc` library is a medium sized library. `glibc` is linked into the runtime image of most hosted software, as well as statically linked into a large number of standalone executables.

Occurrence (5a): Cluster on Diagonal (Pattern 4, Yellow)

Location	Interpretation
Functions 760 to 810	Bottom-up module
Functions 1200 to 1250	Bottom-up module

Inspection of the source-code relation information reveals that at 757 to 815, there are low-level functions concerned with locking, allocating and freeing of memory chunks, and functions up to user-level like `calloc`, `free`, `mcheck`.
Functions 1206 to 1256 contain the elements of a regular expression engine, with functions for handling character elements up to functions for handling strings, and lastly functions for building an acceptance automaton.

Occurrence (5b): Local Horizontal (Pattern 6, Orange)

Location	Interpretation
Function 1300, reference range 50000 to 60000	Module level dispatch function.

The source-code relation shows the function 1306 to be `gaih_inet`. This function implements most of the `getaddressinfo` functionality of the C-library, and therefore has to handle the domain name system (DNS) lookups for internet protocol version 4 (IPv4) and internet protocol version 6 (IPv6) addresses.

5.2.3 Stand-Alone Executable

Embedded systems usually run executables without the help of runtime linkers or hosting environments. Such standalone executables contain all functions from the application and libraries necessary to run. We use a linux kernel as example of such an executable. Note that the plot contains two additional diagonal structures. The first spreads nearly the whole function range, and runs against the direction of the normal diagonal. So far, its

occurrence is observed only for the X86 variant of vmlinux. The second diagonal is at the end of the function range, and in the same direction as the normal diagonal. It corresponds to an instrumentation module or other top-level module that calls into different parts of the kernel. It occurs in the AMD64, X86, and ARM32 variants of vmlinux.

Pattern Example 6: `vmlinux` (see Figure 5.6)
The `vmlinux` executable is a very large executable. `vmlinux` is the kernel of a fully featured personal computer and embedded operating system.

Occurrence (6a): Cluster on Diagonal (Pattern 4, Yellow)

Location	Interpretation
Box structure around function 8700 to 9800 with references between 85000 and 10500	Self-contained module.

Inspecting the names from the source-code relation information, most of the function names between 8834 and 9623 contain the string `ext4_` and make up 96% of the references containing this string in the executable, confirming that there is a module at this position.

Occurrence (6b): Clusters not on Diagonal (Pattern 5, Green)

Location	Interpretation
Functions 10000 to 11100 (references around 360000)	High intermodule coupling.
Functions 22500 to 24000 (references around 310000)	High intermodule coupling.

Inspecting the names from the source-code relation information, most of the function names from 10002 to 11036 contain the string `nfs`. The references that these functions make around 360000 correspond to calls to the functions around 32500, most names of which contain the string `ipv6`. This establishes alink between the network file system (NFS) module and the IPv6 module. The functions between 22418 and 23884 contain the names of multiple drivers of network interface cards. The references these functions make around 310000 correspond to calls to functions around 29000, most names of which contain the string `netdev`.

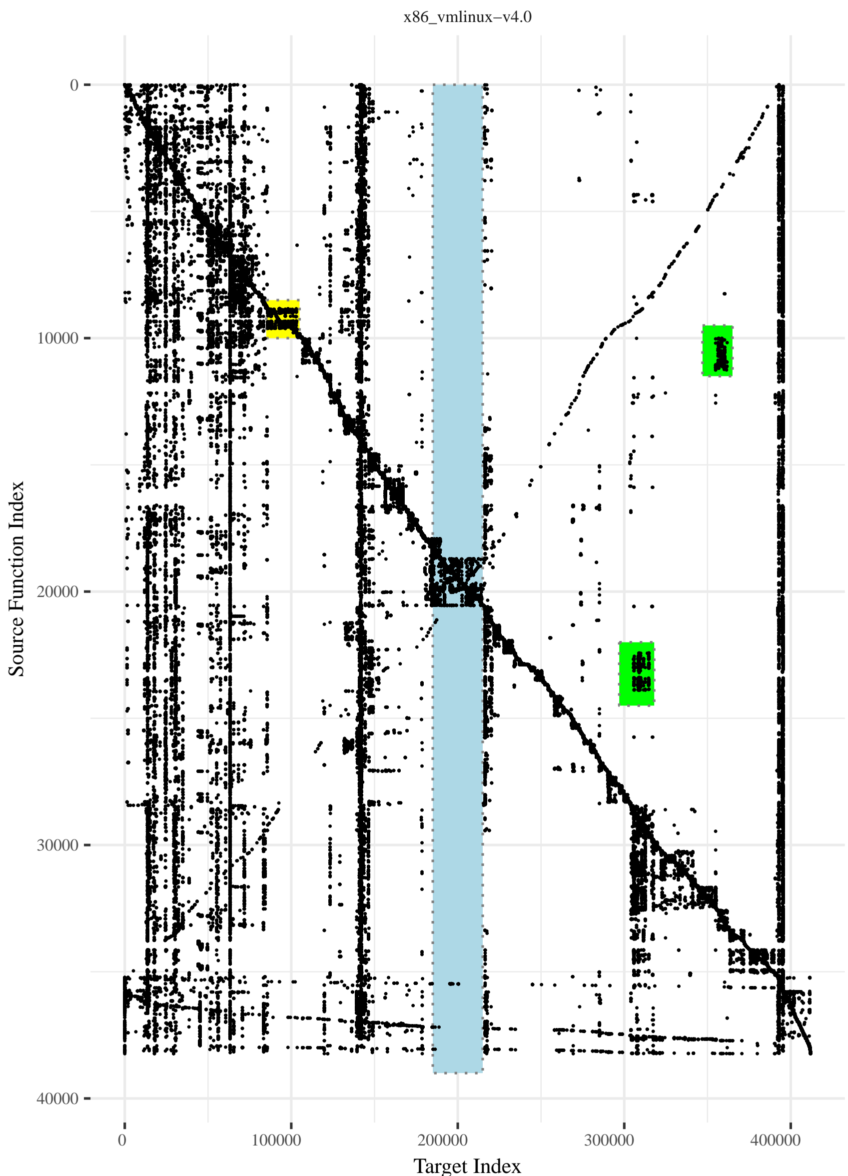


Figure 5.6: Patterns in `vmlinux` Executable

Occurrence (6c): Vertical Space (Pattern 8, Blue)

Location	Interpretation
Nearly no incoming references between references 185000 and 215000 apart from the module cohesion for functions 18000 to 20500	Top-level conglomerate of modules

Inspecting the source-code relation, the functions starting around 18000 contain code of the direct rendering manager, around 18800 of the i9xx driver, and around 19800 to 20573 code concerned with interfaces to displays (high definition multimedia interface (HDMI), low voltage differential signaling (LVDS), video graphics array (VGA)). In this case, the code is one of the top-level modules of the kernel that provide interfaces to the X window system (X11).

As a first conclusion, the application of the patterns to our executable is successful for program comprehension, because the patterns indeed highlight the properties of the control flow for which they are defined, and do so in very different executables.

Next, we investigate the patterns on an embedded executable, where an analysis of large parts of the program is possible.

5.3 Case Study

In this section, we manually apply several of the patterns to a standalone executable and inspect the results in detail. To motivate the following chapter on differential analysis, we additionally inspect the same patterns in another version of the same software. For our analysis, we use the `yaboot` bootloader[1] for the PowerPC architecture used to boot linux (or other) operating systems. It provides access to the hardware of the system, allows interaction in a shell interface, and can load and execute kernel images from the file system.

[1] *Yaboot.* URL: `https://web.archive.org/web/20160312033912/http://yaboot.ozlabs.org/` (visited on 05/19/2017).

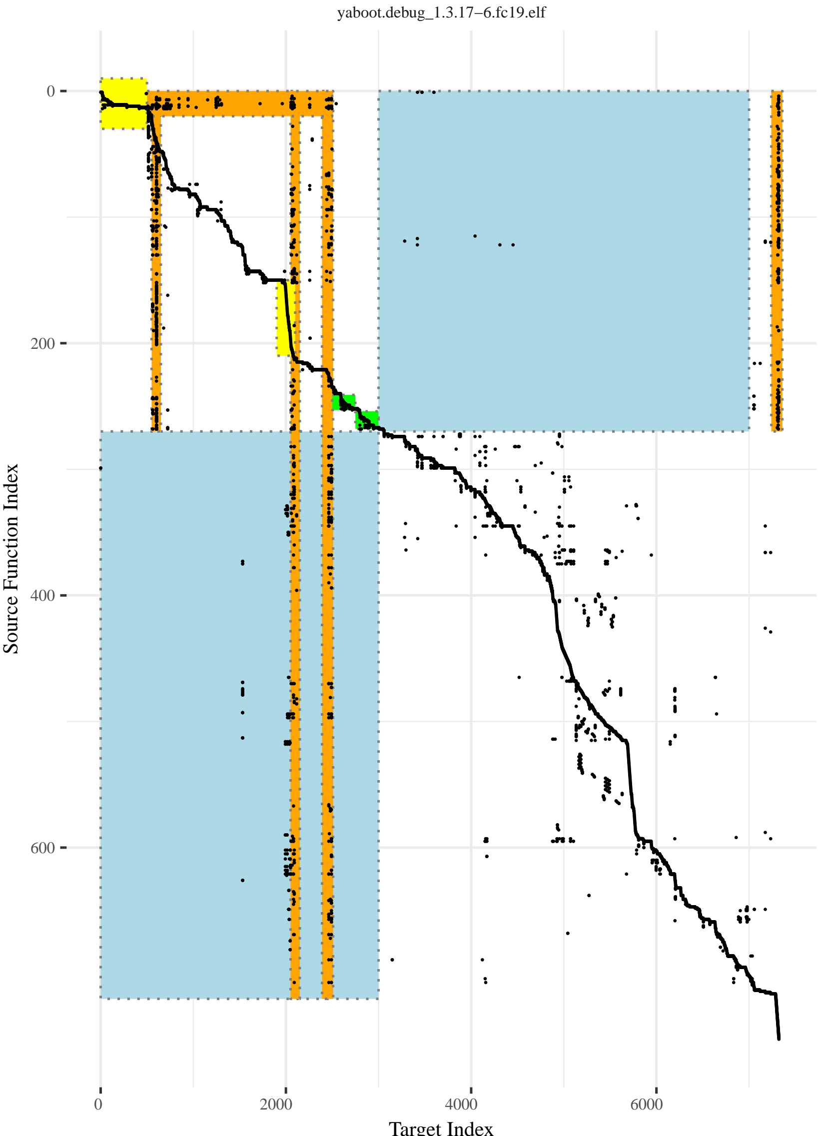


Figure 5.7: Plot of `yaboot` Version 1.3.17

Inspection of a Single Executable First, we perform a detailed inspection of the patterns found in version 1.3.17 of the `yaboot` bootloader as used by Fedora Core 19 (FC19). Figure 5.7 shows a plot of the `yaboot` executable. It contains annotations highlighting several occurrences of our patterns. Again, although suboptimal for the occurrences, we use the same colours to highlight the patterns, as we used in the definition of the patterns. In the following, we describe five groups of occurrences:

1. The plot shows the diagonal line that represents the intraprocedural control flow. The diagonal identifies very big functions (flat part in yellow around source function index 10) and sections of small functions (steep part in yellow around source function index 180±30).

2. The plot shows several larger vertical structures (in orange) around target address indices 600, 2000, 2500, and 7200. These vertical structures mark corresponding functions (intersection of vertical structure with the diagonal) around source function indices 20, 200, 230, and 750 with a high fan-in.

3. The plot shows one horizontal structure (in orange) around the source function index 10 highlighting functions with a high fan-out.

4. The plot has a small number of references (in blue) from sources 0 to 270 to targets 3000 to 7000. Likewise, apart from the vertical structures, there is a small number of references from sources 270 to 750 to targets 0 to 3000. Such an arrangement hints at a partitioning of the executable into two major parts, one from sources 0 to 270, another from sources 270 to 750.

5. The plot has several microstructures (in green) along the diagonal line, for example lower-left triangles around sources 245 and 260. Such microstructures highlight individual modules of the program.

After describing the five groups of occurrences, we investigate each occurrence with respect to the ground truth. Table 5.3 lists the findings made for the `yaboot` executable, together with the ground truth from function names and source filename information, where available.

1. The large functions around function 10 are application-level functions from `yaboot.c`. The small functions around function 180(±30) are small functions from the C library.

Table 5.3: Ground Truth for Patterns in yaboot

#	Pattern	Loc.	Ground Truth
1	Deformed Diagonal	4:13	Top level functions from yaboot.c
1	Deformed Diagonal	180	C-library function
2	Global Vertical	20:42	OpenFirmware library from prom.c
2	Global Vertical	200	String functions
2	Global Vertical	230	Memory functions
2	Global Vertical	750	Save and restore GPR
3	Local Horizontal	4:13	Top-level functions from yaboot.c
4	Vertical Space	0-270	App-level functions (mostly *.c)
4	Horizontal Space	270-750	ext2 library (all *.o)
5	Cluster on Diagonal	241:253	FS module from fs_xfs.c
5	Cluster on Diagonal	254:268	FS module fs_reiserfs.c

2. The sections with high fan-in around 20 are functions from the library that provides a link to the OpenFirmware (like a basic input/output system (BIOS)). The functions around 200 and 230 are the string and memory management functions from the C library, respectively. The functions around 750 implement saving and restoring the general purpose registers (GPRs) of the central processing unit (CPU) and are used to reduce the code size by using these functions in the pro- and epilogues instead of individual implementations.

3. Revisiting the functions around function 10, this time concerning their high fan-out, these functions implement the application-level command line and configuration file interfaces.

4. It is also possible to confirm the separation between application-level code of functions 0 to 275 and library code for functions 275-750. For the first, all but one of the source files have an ending of `.c`, whereas the latter all have a file ending of `.o`, hinting at a different compilation process. This is usually the case for libraries linked in from an archive. The function names identify this library as `libext2fs`.

5. Lastly, the left-lower triangle structures around functions 245 and 260 are additional file-system modules that provide XFS and ReiserFS support.

For each occurrence, the ground truth check did not reveal any inconsistencies.

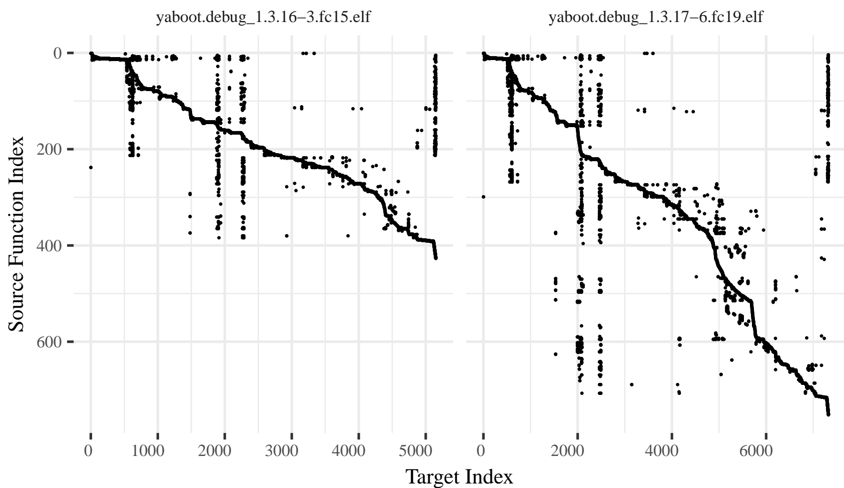


Figure 5.8: Two Versions of `yaboot` from FC15 and FC19

Similarities in Two Versions of the Same Executable In this section, we investigate the similarity of plots for two versions of the `yaboot` executable. Here, in addition to the previous version, we also plot an earlier version, namely 1.3.16 from Fedora Core 15 (FC15).

Figure 5.8 shows two versions of the `yaboot` bootloader. The left plot shows version 1.3.16 from Fedora Code 15, the right plot shots version 1.3.17 from Fedora Core 19. Version 1.3.16 has about 420 functions, version 1.3.17 has about 680 functions. Both of the plots show mostly the same interpretable structures, albeit on translated positions. Table 5.4 lists the positions of the individual findings as previously discussed for the `yaboot` executable. All but one of the findings also present in version .16 are identical to the previous, but the locations of the findings are shifted by inserted functions. Most of the insertions are explained by the different size of the linked in library (`libext2fs`) in the range 270-750 or 200-400, respectively. Much of the rest of the change originates from functions missing from the C-library around function 180±30.

Table 5.4: Locations of Patterns in .16 and .17 `yaboot` Versions

#	Pattern	Location .17	Location .16
1	Deformed Diagonal	10	10
1	Deformed Diagonal	180	–
2	Global Vertical	20	20
2	Global Vertical	200	170
2	Global Vertical	250	180
2	Global Vertical	750	420
3	Local Horizontal	10	10
4	Vertical Space	0-270	0-200
4	Horizontal Space	270-750	200-400
5	Cluster on Diagonal	245	190
5	Cluster on Diagonal	260	210

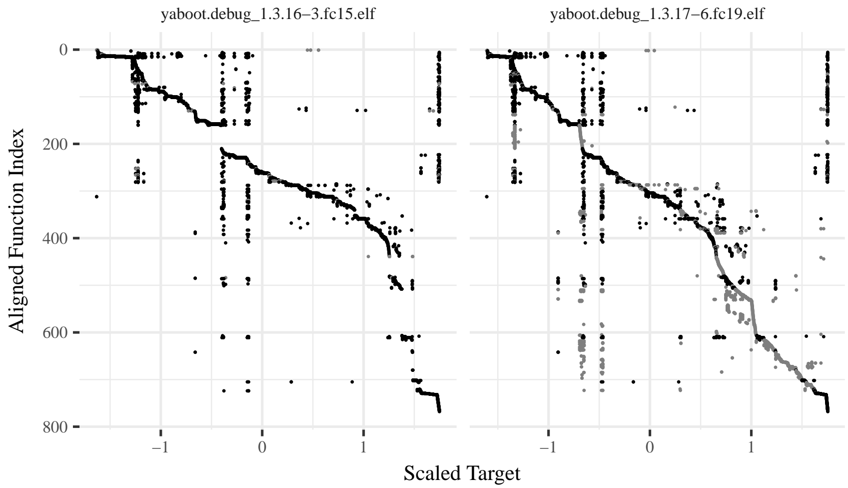


Figure 5.9: Aligned Plot of Two `yaboot` Versions

Aligned Plot of Case Study Figure 5.9 shows an aligned version of the `yaboot` executables produced by aligning (see next chapter) the function names from the source-code relation contained in the debug information. The plot marks functions with identical function names in black and insertions or deletions as grey. Here, the patterns line up visually and any information about one of the two executables can be transferred very easily to the other.

In this section, we have applied the patterns to an embedded executable, and the application of the patterns covered large parts of the executable. The analysis has shown that the information derived from the pattern application was correct with respect to the ground truth, and provided a relatively complete view of the executable: Top-level modules, modules, linked-in libraries, glue code, and helper code can be intuitively identified in this case.

For future work, it would be interesting to automate the search for occurrences of the patterns. A first approach by the author is presented in [47], but the highlighting in this approach is not selective enough. Additionally, an evaluation that reflects whether the orientation eases the program comprehension task, beyond the author using this approach, would be interesting.

6 Differential Analysis

Our differential analysis (DA) addresses analysis cases when more than one executable is available to an analyst. In cases where one executable is already analysed, the DA can aid by allowing a transfer of results from one executable to another. Additionally, the DA can aid by highlighting the similarities and differences between executables and leading further analyses with this information.

In this chapter, we describe the requirements for an algorithm that identifies similarities between pairs of executables or ordered control-flow graphs (OCFGs), respectively. We introduce an alignment technique that meets these requirements using a small example. Next, we formally describe the data representation of our sequence alignments, and introduce several similarity measures for function sizes. We give an intuition about the strengths and drawbacks of the similarity measures using several examples. Lastly, we introduce three measures that estimate the alignment quality.

6.1 Function Size Sequence Alignment

In this section, we discuss the requirements an algorithm for identifying similarities between executables or ordered control-flow graphs (OCFGs) must meet, if it is applied to executables with order-preserving modifications, describe the data available to such an algorithm when applied to executables, and what kind of measure is necessary to estimate the quality of the algorithm's output. We end this section by introducing an alignment algorithm that meets the aforementioned requirements using a toy example.

The algorithm needs to take two OCFGs as input and needs to return a list of function pairs (matching between OCFGs). Secondly, since the matching may be of poor quality, the algorithm needs to return a value that reflects the quality of the matching of the functions in the list.

With such an algorithm, we can perform a differential analysis that serves two purposes: First, it allows transferring the information known about the functions from one executable to another. And second, if both executables are unknown, knowledge can be gained from the variations and similarities of the executables.

We expect the analysis to be precise only for similar executables or similar OCFGs. We can therefore limit our algorithm by making assumptions about similar executables, and by only requesting a useful matching between functions in case of a similar executable. Since a comparison of two non-similar executables of OCFGs results in an alignment where the paired functions are not matches, a quality measure is necessary that predicts the quality of the result. As a side note, such a quality measure then also serves as measure of similarity between executables or OCFGs.

Our simplification consists of assuming the sequence of functions to be stable (see Section 2.4) for the executables our differential analysis is applied to. Therefore, much of the information about the difference is readily available in the function sequence, if the functions were easily comparable. While it is possible to use complex comparisons between the functions, like detailed inspections of each function's control-flow graph (CFG), as it is done by Dullien and Rolles [10], or complex vector-based comparisons on several metrics for each function (e.g., the number of basic blocks, the number of branches, the cyclomatic complexity, the number of calls to other functions, or the number of incoming calls), as it is done by Stojanovic, Radivojevic, and Cvetanovic [40], and also by the author [46], a much simpler measure for each function is possible, if the algorithm, like an alignment algorithm, takes the functions' position in the sequence into account.

A simple comparison of function sizes on its own is a very poor technique to detect similar functions. But, combined with an alignment technique it allows the sequences of function sizes to "snap" into position. Illustratively, one can imagine both sequences of function sizes as a view of a city skyline, with the function sizes being represented as building heights. If, for example, a new skyscraper is built, it is still easily possible to align the other buildings and find the position of the change. The argument is similar for other, more complex changes.

Requirements In this paragraph, we discuss the requirements for a matching algorithm relying on data from disassemblers that is applicable to executables with order-preserving variations. Additionally, we define a ground truth similarity between executables, which our quality measure must reflect.

In Section 2.4, we have discussed several variations in executables that are expected for executables from software product lines. The most important variation is the change in the software revision or version, which implies the addition, deletion, and modification of some or all of the functions. We assume that a major set of functions stays stable in the sequence, when the executable's version changes. The differential analysis (DA) must *represent the addition, deletion, and modification of functions in two function sequences.*

The next probable variation in executables is the modification of compiler options or compiler versions. A change in the optimisation level of an executable usually implies more or less efficient code generation, which changes the amount of instructions needed to perform the same operations. The function sizes can grow due to inlining of other functions. If the compiler performs dead-code elimination, the function sizes can shrink again. The DA must *cope with systematic changes in function sizes in one of the function sequences.*

Secondary to the major expected changes, it is possible that the compiler completely changes, the executables are for two different platforms on the same architecture, or even for completely different architectures, but these changes result in systematic changes in function sizes similar to those discussed above.

The necessary input data for the algorithm depends on two aspects: The first concerns the stability of the function sequence, and the similarity data thereby encoded into this sequence as discussed above, and the second concerns the data available from disassemblers in the general case. The data that disassemblers provide is machine code at various addresses grouped into functions. For this data, the guaranteed measure available for each function is the plain function size. The DA must *use only the function position in the sequence and the function sizes.*

To obtain ground truth about executable similarities, we consider a function as identical in two executables if it is referenced by the same name. For example, the `printf` function present in two executables would be called an identical function. Based on the function names, we give a similarity measure of two function sequences as the ratio of unique function names to all function names. With this information, classical information retrieval measures, such as precision, recall, and F_1 score, can be formulated. It is important to note that for a reverse engineering application, the

Table 6.1: Alignment of A,B,C,D (1,5,1,5) with A,B,C,E (1,3,1,3) using Relative Similarity

(a) Alignment Input and Result

First		Alignment		Second	
V	N	S	M	V	N
1	A	1.0	T	1	A
5	B	0.6	T	3	B
1	C	1.0	T	1	C
5	D	0.6	F	3	E

V: Value, N: Name, S: Sim., M: Match

(b) Similarity Measures

	1	3	1	3
1	1.0	0.3	1.0	0.3
5	0.2	0.6	0.2	0.6
1	1.0	0.3	1.0	0.3
5	0.2	0.6	0.2	0.6

(c) Accumulated Similarities

	–	1	3	1	3
–	**0.0**	0.0	0.0	0.0	0.0
1	0.0	**1.0**	1.0	1.0	1.0
5	0.0	1.0	**1.6**	1.6	1.6
1	0.0	1.0	1.6	**2.6**	2.6
5	0.0	1.0	1.6	2.6	**3.2**

function name information cannot be relied on. We therefore cannot use this comparison for the differential analysis, and have to find other measures to provide an estimate of the alignment quality. The DA must *provide a measure that estimates the alignment quality*.
Note that the evaluation shows that a good measure not only correlates with the achieved quality, but also with the ground truth similarity, which makes this measure a candidate also for a similarity measure between OCFG or executable pairs in the absence of source-code relation information.

Example By applying a Needleman-Wunsch style algorithm for sequence alignment, we can preserve and use the information encoded in the function sequence. Such an algorithm needs to compare functions with each other. Since we only employ the function size as measure for each function, the comparisons only take two numbers each. A more detailed discussion of such comparisons follows in Section 6.3. Such alignment algorithms can optimise a sum of individual comparisons to an optimum, which can be a minimum or maximum. For our introductory example, we use a scoring

that is introduced later as relative similarity: $S(a, b) = 1 - \frac{|a-b|}{max(|a|,|b|)}$ and maximise the sum of similarities.

In Table 6.1a two function sequences are shown, one called First, the other Second. The sequences are very similar in overall shape (small-large-small-large) and only differ in the magnitude of the large parts. One can imagine a normal software development, where the two larger functions are extended. More discussion of this and other examples follows in Section 6.4. For each sequence, the function names are given in sub-columns N and their sizes in sub-columns V. The sequences start with three commonly named functions, and end with one differently named function. Table 6.1b shows the similarity scores between the sequence elements. With this data, the algorithm calculates the maximum sum of scores. In Table 6.1c it is easy to see that for the allowed moves through the matrix (right, down, downright) and related measures (right: 0, down: 0, downright: similarity measure from Table 6.1b), and for the starting point in the top left and destination at bottom right, the best path is along the diagonal, which results in a score of 3.2.

This result is optimal with respect to the names of the functions and matches the first three functions correctly. In this case, the algorithm incorrectly pairs the function D with function E.

In the following sections, we describe a sequence alignment with similarity measures based on function sizes, together with several simple count-based measures, which meet the above requirements.

6.2 Formal Description

This section describes the data preparation, the alignment using Hirschberg's algorithm, and the post processing for the aligned plot representation, as well as the proposed quality estimates.

The alignment is calculated using the function size and position of the function in the sequence as input data.

This input data (see Figure 6.1) is defined as a simplified projection of the `data_c` definition from the ordered control-flow graph (OCFG) (see Section 4.2). Here, we use a set of functions `data_a` that contains the simplified `f_a` and `f_b`, respectively. For each function, we collect the `name`,

```
f_a = (name, start, size)
f_b = (name, start, size)
data_a = ({f_a}, {f_b})
```

Figure 6.1: Sequence Alignment Input Data

```
f_aln = (sim, lname, laddr, lsize,
              rname, raddr, rsize)
data_aln = {f_aln}
```

Figure 6.2: Result of the Alignment Algorithm

`start` address, and `size` in bytes from the original `f_c`. When the similarity score calls for scaled function sizes, we pre-process the data and scale the sizes per executable to zero-mean unit variance using the RobustScaler implementation from SciKit-Learn [30].

To calculate the alignment, we need an algorithm that can optimise (minimise or maximise) a sum of scores and penalties for two sequences. These algorithms are known as optimal alignment algorithm (see Section 3.4), and can usually be parametrised using a scoring, penalties and optimisation goal, which are discussed in the next section. Since the input data can reach significant size very fast, the simple Needleman-Wunsch type algorithms cannot be used, as they require a fully cross-size optimisation matrix in memory. We therefore use Hirschberg's algorithm to calculate the alignment of function sequences, because it uses only $O(min\{n, m\})$ space, and can still calculate the alignment for larger executables like the linux kernel.

The result of the algorithm is a sequence of function pairs with interspaced markers for insertions or deletions. Figure 6.2 shows an alignment result as a set of tuples `{f_aln}`. Each tuple contains the left and right data (prefix `l` or `r`) of the sequence items name `name`, address `addr`, and size `size`. Additionally, each tuple contains the result of the similarity function `sim` used for function comparisons during creation of the alignments. In case of insertions or deletions, the values for one side of the alignment are not defined in the tuple.

The data returned by Hirschberg's algorithm is merged into the previous tuple-based notation of the OCFG. Figure 6.3 shows the extension of the

```
align_data = (seq, match, sim, pos, mode)
AOCFG = { (control_access . align_data) }
```

Figure 6.3: Extension of the Ordered Control-Flow Graph

ordered control-flow graph. Here, each control access tuple `control_access` is concatenated with its respective alignment data `align_data`. The alignment data is derived from the result of the alignment described above. Most importantly, each entry is assigned its position `seq` in the common sequence, the information whether it was correctly matched (`match`) with respect to ground truth source-code relation information, and the similarity `sim` it achieved compared with its corresponding element. Lastly, to be used for the implementation of and as annotations in the graphical views, each element is assigned with an arbitrary position (left or right) in the alignment `pos`, and the method of data preparation for the alignment is saved `mode`.

The most important addition to the source data for the creation of aligned plots is the addition of the position of the individual functions in the sequence. In practice, this results in the assignment of a non-continuous, but strictly monotonic sequence of numbers to each of the sequences, with identical numbers at the positions where Hirschberg's algorithm aligned the functions.

This allows the usage of the `seq` element instead of the `src` or `fnum` elements for the vertical position of a function's control accesses, and therefore directly produces an aligned version of the plot of the function sequences. In addition to the raw sequence data, the plot data is marked `match` for pairs of functions that were aligned, and the similarity score `sim` is saved for later analysis.

Furthermore, simple statistics are calculated to evaluate the alignment quality, both in comparison with a ground truth and without further knowledge.

6.3 Function Size Similarities

It is not easy to choose a good optimisation similarity function for the parametrisation of the algorithm in Section 6.2, since there is no trivially best candidate.

Generally, the algorithm to be parametrised can minimise or maximise a sum of scores for matches, mismatches, insertions, and deletions. For the application case where no function names are available, we use the function sizes as data. Here, we have to manage the trade off between preferring exact matches in the function sizes and handling the variations expected to occur for executables from the same software product line.

The design space of possible aggregated similarity functions (minimisation or maximisation of sum, score functions for matches and mismatches, penalty functions for insertions and deletions) on the whole is infinite. We therefore take a simple constructive approach to explore a part of the space. The evaluation will later show that alignments of useful quality can be calculated with most of the aggregated similarity functions, which leads to the conclusion that the proposed technique is independent of choosing the optimal alignment goal.

The first step is to cope with systematic changes in the function sizes, as they are expected for the variation in compiler settings, architecture, etc. (see Section 2.4). To reduce the impact of these variations, we allow the scaling to zero-mean unit variance of the function sizes. In the second step, on this optionally scaled data, we define 5 comparison strategies, 4 of which use a maximum sum of similarities, and the other uses a minimised sum of differences and penalties. The design requirements for the similarity functions are as follows: Firstly, the function must return 1 for identical function sizes. Secondly, to cope with functions of differing sizes due to changes in version, changes in function sizes should be tolerated. Lastly, for alignments with identical compiler and optimisation, identically sized functions are very likely to be equal and should be preferred.

All functions compare elements a and b from the whole sets of elements A and B. For the similarity functions all insertions and deletions (comparisons of a single element with nothing) are valued zero.

Similarity Measure 1 (Inverse Similarity)

$$InvSim(a, b) = \frac{1}{1 + |a - b|}$$

The *inverse similarity* function is a similarity function that returns 1 for identical parameters, and decays to 0 for growing differences in the parameters.

Similarity Measure 2 (Exponential Similarity)

$$ExpSim(a, b) = 2^{-|a-b|}$$

The *exponential similarity* function is another similarity function that returns 1 for identical parameters, and decays exponentially to 0 for growing differences in the parameters.

Similarity Measure 3 (Maximum Similarity)

$$MSim(a, b) = 1 - \frac{|a - b|}{M}$$

where $M = |max_elem(A \cup B) - min_elem(A \cup B)|$

The *maximum similarity* function is a similarity function that returns 1 for identical parameters, and decays linearly to 0 for growing differences in the parameters. 0 is reached for the maximum possible difference in function size between the unions of the sets of function sizes of the executables.

Similarity Measure 4 (Relative Similarity)

$$RSim(a, b) = 1 - \frac{|a - b|}{L}$$

where $L = max(|a|, |b|)$, special case: $RSim(0, 0) = 1$

The *relative similarity* function is a similarity function that returns 1 for identical parameters. For parameters where one parameter is small and the other parameter is large, it behaves like the inverse or exponential similarity functions. For parameters of similar magnitude, it behaves like the MSim similarity function. For scaled function sizes that also reach negative numbers, the function returns -1 for parameters of the same magnitude but of differing signs. Returning negative numbers is no problem for subsequent measures depending on the similarity values, since these only request the similarity functions to have a maximum value of 1, so as not to exceed a counter on the same pairs.

Figure 6.4 shows a comparison of the four similarity functions on unscaled function sizes. Note that the evaluation of the functions starts at 1, since

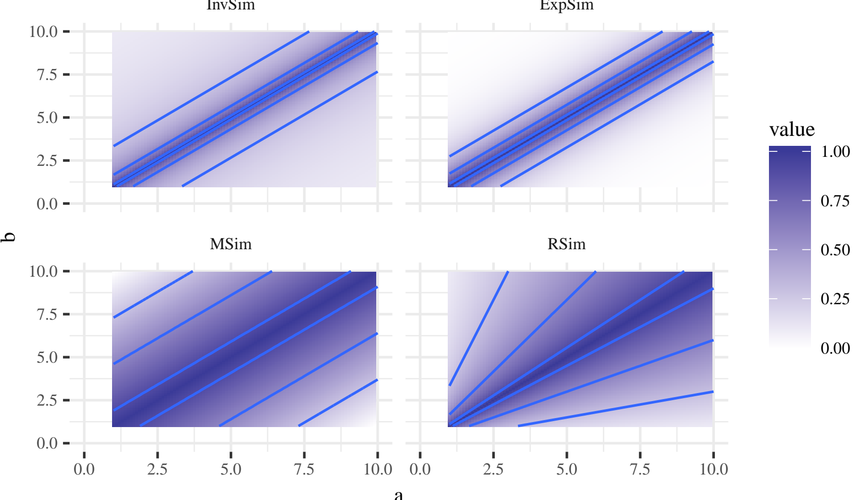


Figure 6.4: Comparison of the Similarity Function on Unscaled Function Sizes

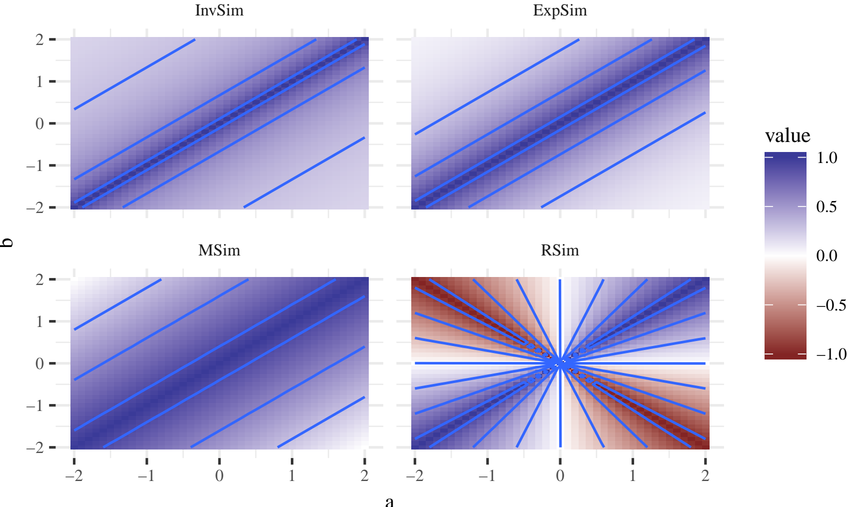


Figure 6.5: Comparison of the Similarity Function on Scaled Function Sizes

unscaled function sizes are never valued 0. The InvSim and ExpSim functions show that they rapidly decay for differences in their parameters. Both functions mostly address the design idea of preferring identically sized functions. The MSim function shows the linear decay with evenly spaced contour lines. It relates to the design idea of tolerating changes in the function sizes. The RSim function shows complex behaviour mixed between the linear and non-linear functions. It discourages the alignment of small functions, except if they have identical sizes.

Figure 6.5 shows a comparison of the four similarity functions on scaled function sizes. The InvSim, ExpSim, and MSim functions behave identically on the scaled data and on the unscaled data. The RSim function changes its behaviour of discouragement to the alignment of average-sized functions. It now also penalises the alignment of functions with different signs. With this behaviour, it snaps the alignment on the extreme values in the function sizes.

Since the optimisation goal of Hirschberg's algorithm can also be defined as minimising a sum, we also provide a function not for measuring similarities,

but for differences. In the following, to limit confusion, and since this function on the whole behaves like a similarity function, we also call it a similarity function.

Similarity Measure 5 (Size Difference)

$$Diff(a, b) = |a - b|$$

$$Pen(i) = |i| + \epsilon$$

Here, matches are valued 0 (result of Diff with equal parameters) and mismatches according to the absolute value of their values difference (Diff). Contrary to the above maximising functions, insertions and deletions have to be penalised (Pen), at least as much as their alignment with a zero element. To additionally prefer the alignment of zero elements with other zero elements (scaled elements), we add a very small value (ϵ) to the penalty.[1]

Selecting the best similarity function for a given alignment is not trivial. We therefore investigate the possible choices in each of the following case studies. To aid in choosing a similarity function, we give a summary of the alignment qualities in the case studies by similarity function in Section 7.1.3, and provide an intuition about each of the similarity functions performance on simple examples in the next section.

Note that in the following, we use aggregated similarity function and similarity function synonymously, since these are given as a one-to-one relation.

6.4 Fully Worked Examples

To provide a feeling of the alignments produced by different similarity functions, we investigate them with several small examples.

Tables 6.2 to 6.3 show the toy examples used to test the similarity functions, both in the unscaled and in the scaled representation. Each example consists of two number sequences, a rough description of what these sequences would describe if they were function sizes of software, and a categorisation into the expected variations.

[1] The implementation uses the floating point machine epsilon.

Table 6.2: Characteristics of Example Sequence Pairs

#	Element	Version	Compiler	Optimisation	Platform	Architecture	Code
1	□	■	□	■	□	□	Normal Development
2	▒	■	□	□	■	□	Shared Module
3	▒	▒	■	□	□	□	Dissimilar; Permuted Software; Shared Module
4	▒	▒	▒	■	▒	■	Dissimilar; Architecture Change
5	■	■	□	□	▒	▒	Shared Module

■: Changes, □: No Changes, ▒: Possible Changes

Table 6.3: Example Sequence Pairs

(a) Unscaled

#	Sequence A	Sequence B
1	1, 5, 1, 5	1, 3, 1, 3
2	1, 2, 3, 4	3, 4, 5, 6
3	1, 2, 3, 4, 5, 6	5, 6, 4, 3, 2, 1
4	1, 2, 3, 4, 5, 6, 7	1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7
5	1.1, 1.2, 1.3, 5, 6, 7	5, 6, 7, 1.2, 1.3, 1.4

(b) Scaled

#	Scaled Sequence A	Scaled Sequence B
1	-1, 1, -1, 1	-1, 1, -1, 1
2	-1.3, -0.4, 0.4, 1.3	-1.3, -0.4, 0.4, 1.3
3	-1.5, -0.9, -0.3, 0.3, 0.9, 1.5	0.9, 1.5, 0.3, -0.3, -0.9, -1.5
4	-1.5, -1, -0.5, 0, 0.5, 1, 1.5	-1.5, -1, -0.5, 0, 0.5, 1, 1.5
5	-1.01, -0.97, -0.93, 0.57, 0.97, 1.38	0.56, 0.97, 1.38, -1.01, -0.97, -0.93

The first example represents functions with small changes, as would be expected for small changes because of version changes, or changes in compiler flags. The second example represents an unchanged shared module between two executables of different platforms or versions. The third example represents two different or permuted executables with a small shared module, as would be produced by a change of compiler and compilation process. The fourth example represents an artificial case, as changes as drastically are not realistic, but similar behaviour can be seen for changes in compiler or architecture. Still, this example is interesting in the context of scaled function sizes, as it is produced by scaling and offsetting itself (see Table 6.3). The last example, again, represents the case of a shared module, as would be expected for a larger version change. This example is deliberately modelled to be very sensitive as to provide a possible alignment partner of both parts of the sequence.

In the scaled representation of the sequence pairs, the first, second and fourth example have identical values per sequence pair. The third example has values of identical magnitude but opposing signs in the sequences. In the last example, the formerly identical part of the sequence is now scaled to more different values (different in the second digit behind the decimal point), and the formerly different part is scaled to more similar values (no difference in the visible parts; only differences in the third digit behind the decimal point).

Table 6.4 show alignments of all example sequences for all similarity functions without and with scaling. The first table shows all unscaled alignments with one example per row. Each alignment shows both sequences and the result value of the optimisation. The similarity functions are shown in the columns. They are grouped by minimisation or maximisation of the sum and by the comparison function. In the second table, all scaled alignments are shown using the same layout. Each example alignment uses a box notation for the pairs in the alignment, where ▫ marks an insertion, ▫ marks a deletion, ■ marks a pair with identical values, and ▪ marks a pair with different values. Note that neither pairs with identical values nor pairs with different values are necessarily correct.

For each of the examples, we discuss the resulting alignments, together with their optimisation score and its potential to be interpreted as a measure of the validity of the alignment result. The results of the minimisation are shown for completeness only and are not discussed in detail, in particular

Table 6.4: Example Sequence Alignments using all Optimisation Goal Candidates

(a) Unscaled

	Max				Min
#	InvSim	ExpSim	MSim	RSim	Diff/Pen
1	2.7	2.5	3	3.2	(4)
2	2	2	2.4	2.2	(8)
3	2.1	2	2.6	2.8	(14)
4	2.7	2.2	3.9	3.1	(18)
5	3	3	3	3	(7.5)

: Insertion, : Deletion, : Identical Values, and : Different Values.

(b) Scaled

	Max				Min
#	InvSim	ExpSim	MSim	RSim	Diff/Pen
1	4	4	4	4	(0)
2	4	4	4	4	(0)
3	2.5	2.3	2.6	2	(5.9)
4	7	7	7	7	(~0)
5	3	3	3	3	(5.8)

: Insertion, : Deletion, : Identical Values, and : Different Values.

the numeric result of the alignment score is not directly usable as part of a quality measure.

The first example represents the alignment of two similar sequences of functions. All similarity functions produce an alignment of length four, in which all elements are paired up. Considering the result values of the optimisations, all scaled maximisations have reached the maximum value of 4, since the scaled sequences are equal. The unscaled function size alignments show values between 2.5 and 3.2. The assumption of version or optimisation changes suggests a correct value of 4 similar pairs. The *relative similarity* function's result is closest to this value with a score of 3.2.

The second example represents an alignment of two sequences where the suffix of the first is the prefix of the second. The similarity functions produce three types of results: One, where the two last elements of the first sequence are aligned with the first two of the second; the second, where the last three elements of the first are aligned with the first three of the second; and the third, where all elements are aligned. Assuming a shared sub-module, the alignments with two pairs produced by *inverse similarity* and *exponential similarity* on unscaled sizes are correct alignments with a value of two. These hint at the tendency of these alignments to prefer exact matches in size. The other two results on unscaled sizes align three pairs with values of 2.4 and 2.2. These show the tendency of the *maximum* and *relative similarity* to align similarly shaped parts of the sequence without the need of a prescaling. All of the scaled alignments as well as the minimisation alignments have four aligned pairs. For the scaled alignments this is trivial, since the scaled sequences are identical.

The third example again represents an alignment of two sequences where the suffix of the first is the prefix of the second, but here, the non-similar parts of the sequences are permuted, and the sequences have a different general shape. The similarity functions produce three types of results: The first, where the elements with value six (sixth of the first sequence and second of the second sequence) are not aligned; the second, where the elements with value one (first of the first sequence and last of the second sequence) are not aligned; and the third where the elements with values five and six of both sequences are aligned with each other. Assuming a shared sub-module of two functions, the alignments produced by the unscaled *exponential similarity*, scaled *relative similarity*, and scaled minimum difference goals are correct (with value 2). The other alignments only separated out the

extreme elements (1 or 6) and otherwise aligned the remaining elements. This confirms that the alignment technique always produces some alignment, which also possibly aligns the noise part of the signal.

The fourth example represents an alignment of a sequence with a scaled and shifted version of itself. All except one similarity function produce the same result of a full alignment.
Assuming large changes in optimisation settings (each element is scaled and an offset is added, but both sequences represent identical elements), most alignments performed well, with the alignments with scaling producing a full value of 7 (the sequences are identical after scaling). The unscaled alignments produce the correct result by chance, as the example of the *exponential similarity* function shows, which without apparent reason left the elements with values five and six of the first sequence (last two elements) and the elements of value 1.2 and 1.3 of the second sequence (second and third element) unpaired.

The last example represents an alignment of two sequences that are split into two parts each, where the first part of the first sequence is similar in shape and magnitude to the second part of the second sequence, and the second and first part, respectively, are identical. Here, the comparison functions on unscaled sequences all produce alignments of the identical parts, and the scaled versions produce alignments of the similar parts. This example shows that the alignment technique can be very sensitive to unlucky data preparation or choice of similarity function.

In summary, the examples show that there is no one size fits all similarity function, and the selection needs to be done with knowledge of the expected variation in each case, since otherwise, low quality alignments are possible. There are two mitigations to this problem. One is the inspection of the aligned plots of an alignment, where the patterns discussed in Chapter 5 should show up aligned with each other. The second is the estimation of the alignment quality by a measure based on the alignment score. Additionally, the statistics discussed later about the similarity measures (see Section 7.1.3) show that there usually are other similarity measures which outperform the first choice.

6.5 Quality Estimates

In the domain of bioinformatics, measures of alignment quality are commonly based on the sequence identity, that is, the ratio of exactly paired sequence elements compared to the total number of elements. Such a simple counting measure is generally not available on aligned function sizes, as we expect systematic changes in the sizes, for example, due to function inlining. We therefore cannot use an exact identity function. But, we can reuse the similarity score of the alignment algorithm for the cases where the alignment calculates a maximum sum of similarities. This score also aggregates one (1) values for the similarities in case of identical sequence elements, and smaller than one values for differing sequences elements. By relating the pairwise sum of this score to the length of the common subsequence, the minimum necessary length, or the length of the whole alignment (with insertions or deletions), a measure can be constructed, which is similar to the sequence identity.

For example, recalling the example from Section 6.1, where the sequences 1,5,1,5 and 1,3,1,3 are aligned, the resulting similarity score was 3.2. This value in relation to the alignment length of 4 is 3.2/4=0.8, which can be interpreted as a similarity of 80% of the alignment. Many more examples of these measures are given in Section 7.1.1, and the evaluation contains a part specifically concerned with these measures in Section 7.3.

Table 6.5 shows variables based on the alignment data described in Section 6.2. There is a fixed order between the variables: The sum of similarities *sum(sim)*, for similarity functions that have an upper bound of one, is necessarily always smaller or equal to the number of pairs *len* in the alignment. The number of pairs can only reach the minimum length *minlen* of both executables, which is by definition smaller or equal to the maximum length *maxlen*. The aligned length *alignlen* is always larger or equal to this maximum length, because all functions of the larger executable must also be included in the alignment.

Next, we define three related measures based on the variables defined above, to provide information about the alignment quality. All three measures use $sum(sim)$ as approximation to the number of matched functions. For identical executables, this sum is essentially a count of the paired functions (all similarities *sim* equal to one). To provide an interpretable measure, this

Table 6.5: Variables Based on the Alignment Data

Count	Description
sum(sim)	The sum of the individual similarities for all aligned pairs. This is the value that is maximised by the alignment algorithm.
len	The length of the longest common subsequence in the alignment between the executables, i.e., the total number of function pairs.
minlen	The minimum of the number of functions for both executables.
maxlen	The maximum of the number of functions for both executables.
alignlen	The length of the whole alignment, including insertions and deletions, e.g., the pairs of functions, where one function is blank.
	sum(sim) $\leq$ len $\leq$ minlen $\leq$ maxlen $\leq$ alignlen

sum has to be related to a potential maximum score. Note that we use percent notation of the metrics in the evaluation.

Similar to how precision and recall are defined to assess the ground truth of the alignments (see Equations (7.2) and (7.3)), which are relations of the number of correctly matched elements to the length of the alignment or the number of existing identical elements, respectively, the first candidate relates the sum of similarities to the length of the alignment *len*. This leads to the first definition of *match similarity*.

$$\text{match similarity} = \frac{\text{sum(sim)}}{\text{len}} \tag{6.1}$$

But, as the evaluation shows (see Section 7.3), this measure has problems to properly represent alignments of low quality, that is, in the extreme it is high ($> 80\%$) for both low ($< 20\%$) and high ($> 80\%$) F_1 scores. Intuitively, this behaviour relates to the alignments of noise as discussed for the third case of the above examples.

The *match similarity* can be interpreted as a measure of how good the

optimisation goal of the alignment (maximise the sum of similarities for all pairs) was met, with the achieved alignment.

To mitigate this noise problem, a measure is needed that is larger than *len* for low quality alignments. The next variable larger than *len* is *minlen*, which represents the length of the shorter sequence and therefore the maximum possible length of a common subsequence. With it, we define the *score ratio* measure.

$$\text{score ratio} = \frac{\text{sum(sim)}}{\text{minlen}} \tag{6.2}$$

As the evaluation (see Section 7.3) shows, this measure correlates better with the F_1 score than *match similarity*, but still has the same problem of giving low scores to the alignments of very low quality for different executables. The *score ratio* relates the optimisation goal to the maximum achievable goal, assuming that one executable's functions are a subset of the other executable.

The largest of the above measures is *alignlen*, which is the length of the whole alignment. With this measure, we define the *alignment similarity*.

$$\text{alignment similarity} = \frac{\text{sum(sim)}}{\text{alignlen}} \tag{6.3}$$

The evaluation shows (see Section 7.3) that this measure is able to provide a useful correlation to the F_1 scores and is therefore useful for estimating the quality of the alignment. A threat to the validity of this measure is that in our test suite, the quality of the alignment also inversely correlates with the difference in sequence lengths of the input sequences.

7 Evaluation

This chapter begins with a description of the setup of the evaluations, namely how we obtain ground truth, how the test suite is constructed, which similarity functions we evaluate, and how a canonical case study looks. Then follows the actual evaluation data for the different test cases, and a summary of the evaluation in terms of quality.

In the evaluation, we cover the following questions: *To which degree are the executable variations order-preserving?* This question concerns both the structural as well as the differential analysis. For the structural analysis, it can ensure to which degree the patterns can be applied without fear of the executable being reordered in the build process. For the differential analysis, it tells whether the sequence alignment approach is applicable. We investigate this question by inspecting the correlation of similarity and stability of the function sequences of the executable pairs.

To which degree does the differential analysis work for the order-preserving variations? Or, another way to ask, is the alignment technique effective with respect to the potential of the input? We investigate these question by inspecting the similarity versus the achieved quality of the function sequence alignments of the executable pairs.

To which degree can the differential analysis estimate the quality of the returned alignment? We investigate this question by inspecting the quality estimates versus the achieved quality of the function sequence alignments of the executable pairs.

7.1 Setup

In this section, we describe the evaluation setup. First, we describe how ground truth about the executable similarity and stability is obtained. Secondly, we describe the construction of the testsuite. Thirdly, we describe how the parametrisation of the alignment algorithm used for the evaluation is selected. And lastly, we describe the data presentation and common layout used in the canonical descriptions of each test case.

Table 7.1: Counts of Identical Function Names in Executables and Alignments

Count	Description
identical	The number of functions with identical names in the aligned executables, i.e., the size of the intersection between the sets of function names that do not start with `sub_`.
unique	The number of unique function names in the aligned executables, i.e., the size of the union between the sets of function names that to not start with `sub_`.
match	The number of function pairs in the alignment with identical names that do not begin with the characters `sub_`.
	match ≤ identical ≤ unique

7.1.1 Ground Truth

We use function names from the source-code relation information to inspect the similarity and stability of the function sequences in the test suite, and to evaluate the quality of the alignments.

In Table 7.1, we define two function counts based on comparisons of the function names in both unaligned sequences, and one count based on the pairs of aligned sequences. These counts are used to calculate measures of the true quality of the alignment.

We have to filter the function names with the prefix `sub_`, since these are auto generated by IDA Pro, when the source-code relation information of an executable does not contain the function name, and it cannot be guessed otherwise.

The *identical* and *unique* counts simply represent the number of identically named functions between the executables, and the whole set of function names in both executables, respectively. The *match* count represents the number of correctly paired functions in an alignment.

There is a trivial ordering between the counts as follows: *match* ≤ *identical* ≤ *unique*. The number of matched function names can never be greater than the number of identical function names, which itself cannot be greater than the number of unique function names.

Since the differential analysis mainly depends on the idea of similar executables and the assumption that similar executables have stable function sequences, these aspects have to be evaluated for the elements of the test suite.

$$\text{similar} = \frac{\text{identical}}{\text{unique}} \tag{7.1}$$

We provide a simple measure of the similarity (*similar*, see Equation (7.1)) between two executables, by dividing the number of identically named elements *identical* by the number of unique element names (*unique*). For two executables with the same set of function names, this measure results in 1, for executables without shared function names, this measure results in 0. Note that the similarity is not compensated for effects of executables that have large differences in the number of functions.

To inspect the stability of the alignment, we calculate an alignment with a similarity function that performs comparisons based on function names, and returns 1 if the names are identical and 0 otherwise. This will result in an ideally precise alignment (all sequence elements names are always matched).

$$\text{recall} = \frac{\text{match}}{\text{identical}} \tag{7.2}$$

The *recall* (see Equation (7.2)) of this alignment, which is the ratio between the number of matched function names *match* and the number of identical function names *identical*, provides a check of the assumption of sequence stability. Generally, the *recall* is a statistical measure which reflects on the true positives compared to the correct number of positives. For executables with identical function sequences (identically named functions, in the same order), the number of matched function names equals the number of identical function names, and the measure returns one. If the functions are all identically named, but not in the same order, the alignment calculates the maximum subset of function names that are in the same order. In this case, the recall indicates the ratio of ordered functions to the whole function set. If the function sets only overlap and each set has unique elements, the recall represents the ratio between the maximum set of ordered elements in the join of the sets to the size of the join of the sets.

Overall, the *similar* and *recall* measures with alignments on function names allow the required inspection of the test suite.

To assess the achieved quality of alignments based on function sizes, we use the F_1 score. This commonly used score is a statistical score which combines the recall (as defined above) and the precision (defined next) into a single score.

$$\text{precision} = \frac{\text{match}}{\text{len}} \tag{7.3}$$

The *precision* (see Equation (7.3)) is the number of matched functions *match* divided by the length of the common subsequence *len*. Generally, it is the ratio of correct elements in the result compared with all elements of the result. For alignments where all elements of the longest common subsequence are correctly matched (w.r.t. function names), the precision measure's value is 1. If the longest common subsequence does not contain correctly matched elements, the measure's value is 0.

Since both the *recall*, which represents the ability of our differential analysis (DA) to find elements, and the *precision*, which represents the ability of our DA to return relevant elements, are very important in our use cases, we combine the measures with a harmonic mean (commonly defined as the F_1 measure and related to the effectiveness measure for information retrieval [32]).

$$F_1 = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{7.4}$$

This combination returns a value of 1 if both precision and recall are valued 1, and a value of 0 if any of precision or recall are valued 0. This limits the effects of cases where a single common element is by chance correctly paired ($recall = \frac{1}{1} = 1$) between two almost disjoint sets of elements ($precision = \frac{1}{N} \approx 0$).

Generally, we use the F_1 score as approximation of the achieved quality of our alignments based on function sizes. Note that we use percent notation of the metrics in the evaluation.

Quality Measures Example Revisiting our example of `yaboot` executables from Section 5.3, we have a first look at the measures calculated for individual alignments.

Table 7.2 gives the detailed numbers of the input data. The two files in the alignment have 427 (*minlen*) and 752 (*maxlen*) functions, respectively. The

Table 7.2: Common Measures Calculated for the Alignment of `yaboot` Executables

Measure	Common
fileA	yaboot.debug_1.3.16-3.fc15.elf
fileB	yaboot.debug_1.3.17-6.fc19.elf
identical	415
len(fileA) = minlen	427
len(fileB) = maxlen	752
unique	764
similar	415/764 = 54%

source-code relation information shows 415 names (*identical*) that appear in both of the executables out of a total of 764 names (*unique*). This results in a *similar* measure of 0.54, which means that 54% of the known names appear in both executables.

Table 7.3 gives the detailed numbers of the calculated alignments. The table contains the measures calculated for each of the different similarity functions in separate columns.

The first data column shows the alignment calculated on the identity of function names. The common subsequence (*len*) here contains 411 elements, all of which have identical names (score 1). The precision of this result is as expected 100%. Relating these 411 elements (*match*) to the 415 elements total (*identical*) results in a recall of 99%. In this case, almost all of the elements present in both files appear in the exact same order.

The best alignment quality (highest number of correctly matched functions) is produced using the unscaled relative similarity (RSim). The number of elements that the alignment returned as pairs of functions (*len*) is 421. With the names from the source-code relation information, 356 of these are determined to be correct pairs (*match*). The sum of the individual similarities is 397 (*sum(sim)*).

These numbers result in a *precision* of 85% and *recall* of 86%, which means that 85% of the function pairs were correctly aligned and 86% of the total number of identical functions were returned. In terms of F_1 score, this results in 85%.

Table 7.3: Measures Calculated for the Alignment of `yaboot` Executables

(a) Unscaled

Measure	Name	InvSim	ExpSim	MSim	RSim	Diff
match	411	336	303	318	356	307
sum(sim)	411	238	224	424	397	*61044*
len(LCS)	411	403	365	426	421	423
alignlen	768	776	814	753	758	756
precision	100%	83%	83%	75%	85%	73%
recall	99%	81%	73%	77%	86%	74%
F_1	100%	82%	78%	76%	85%	73%
match sim.	100%	59%	61%	99%	94%	–
score ratio	96%	56%	52%	99%	93%	–
alignment sim.	54%	31%	28%	56%	52%	–

(b) Scaled

Measure	InvSim	ExpSim	MSim	RSim	Diff
match	292	238	222	293	309
sum(sim)	352	358	419	300	*365*
len(LCS)	424	423	426	398	418
alignlen	755	756	753	781	761
precision	69%	56%	52%	74%	74%
recall	70%	57%	53%	71%	74%
F_1	70%	57%	53%	72%	74%
match sim.	83%	85%	98%	75%	–
score ratio	83%	84%	98%	70%	–
alignment sim.	47%	47%	56%	38%	–

Next, we inspect the proposed quality estimates of the unscaled relative similarity: The *match similarity* for the aligned functions is 94%. The *score ratio* is 93%, which means that 93% of the possible maximum alignment score (*minlen*) of 427 was reached in the calculated alignment. The *alignment similarity* is 52%.

For the other goals, the returned alignments are not individually discussed. All of the returned alignments show that the *precision* and *recall* are similar, and their difference is always smaller than 10% points. This allows the F_1 scores to be used as a representative summary of both of these measures in the following evaluation.

The table also lists the results for the *size difference* alignments. These values are included for completeness. Note that the sum of similarities *sum(sim)* is the value to be minimised in this case, and especially for the unscaled function size, has a difference in magnitude compared to the other values.

Note that each of the calculated alignments uses a different similarity function, which prohibits a direct comparison of the quality estimates, since these are based on the sum of these similarities. The prediction quality of the quality estimates is assessed for a single similarity function in the evaluation in the following section.

7.1.2 Test Suite Construction

In this section, we discuss the design space of a test suite for our evaluation.

The evaluation of the DA mainly needs to assess alignments calculated on similar executables. With this, it is possible to inspect the stability of function sequences and the performance of the proposed method. Additionally, an inspection of dissimilar executables helps to assess the behaviour of the proposed method in corner cases. A second dimension is the environment in which the executable normally runs. Here, we mainly differentiate between libraries, which are part of other executables, standalone executables, which run without interaction with other code, and hosted executables, which interact with dynamically linked libraries as well as operating systems. The hosted executables and libraries cover most of the population of general executables, and the standalone executables cover embedded executables. Lastly, we use executables from a third party [40] that were used in the

Table 7.4: Evaluation Suite: Similar Executables

Entity	Category	Variation	# Executables
The GNU C Library (glibc). URL: `http://www.gnu.org/software/libc/` (visited on 05/19/2017)			
`glibc`	Compiler Optimisation Architecture Version	gcc -O1,2,3,s AMD64 2.17 - 2.21	18
musl libc. URL: `http://www.musl-libc.org/` (visited on 05/19/2017)			
`musl`	Compiler Optimisation Architecture Version	gcc, clang -O0,3 AMD64 0.9.0 - 1.1.10	68
RedBoot. URL: `https://sourceware.org/redboot/` (visited on 05/19/2017)			
`redboot`	Architecture Platforms	ARM, MIPS, PowerPC, X86 *various*	28
Linux Kernel Source Tree. URL: `https://github.com/torvalds/linux` (visited on 05/19/2017)			
`vmlinux`	Compiler Architecture Version	gcc AMD64, ARM, X86 3.0 - 4.0	63

evaluation of a different analysis that finds functions similar to a given function from a set of other functions (one to N queries, not N to M queries as in our case). This allows us to validate our results against executables which were not directly self-selected.

Tables 7.4 to 7.6 list the entities of the test suite and their variation parameters, as well as the number of elements used in the evaluation for each test case. The first table contains similar libraries and standalone executables, the second table a set of dissimilar hosted executables, and the last contains two sets of executables from a third party. All, except the

Table 7.5: Evaluation Suite: Dissimilar Executables

Entity	Category	Variation	# Elements
SPEC CPU^{TM} 2006. URL: `https://www.spec.org/cpu2006/` (visited on 05/19/2017)			
`CPU2006`	Elements	*various*	20
	Compiler	gcc	
	Optimisation	-O2	
	Architecture	X86	

dissimilar entries, are present in different variations. These variations are chosen to meet the variations described in Section 2.4. We obtained `redboot`, `stamp`, and `bb_matrixssl` executables as binaries. The other executables are built from source to provide a wider span of the variations as can be obtained from prebuilt real-world executables. The `redboot` executables were directly obtained from the `redboot` website at the time of writing. The `stamp` and `busybox matrixssl` executables were obtained from Stojanovic, Radivojevic, and Cvetanovic [40]. The self-compiled members of the test suite are chosen to provide a good possibility of influence to the build process.

The `glibc` and `musl` libraries are C-libraries that provide low-level functions for a C program as well as interfaces to the hosting environment. These libraries have hundreds of functions each. The `redboot` bootloader is a standalone embedded bootloader, which provides basic network and file system functionality and can load and start an operating system. These executables also have hundreds of functions each. The `vmlinux` kernel is a standalone operating system kernel that provides the base of a feature-rich multitasking environment with many hardware drivers. These executables have between ten- and thirty-thousand functions.

The `CPU2006` benchmark is a set of hosted executables, which are instrumented so they can be used to assess the performance of the compilers and processors of a system on the whole. These executables have widely varying function counts.

The `stamp` benchmark and `busybox matrixssl` executables stem from an evaluation of a feature vector based differential analysis of executables, where a single function is queried against a database of functions. The

Table 7.6: Evaluation Suite: Validation Executables

Entity	Category	Variation	# Elements
Sasa Stojanovic, Zaharije Radivojevic, and Milos Cvetanovic. "Approach for Estimating Similarity between Procedures in Differently Compiled Binaries". In: *Information & Software Technology* 58 (2015), pp. 259–271			
`stamp`	Elements	*various*	50
	Compiler	CS, CW, IAR, KAIL, SYS	
	Optimisation	-O0,3	
	Architecture	ARM	
`Busybox`	Compiler	CS, CW, IAR, KAIL, SYS	14
`matrixssl`	Optimisation	-O0,3,s	
	Architecture	ARM	

compilers used in the creation of these executables are: CS: CodeSourcery ARM GNU/Linux tool chain, CW: CodeWarrior GCC for ARM, IAR: IAR Embedded Workbench, Keil: Keil ARM Compiler, SYS: SysProgs Prebuilt GNU Toolchain.

In summary, the test suite consists of similar and dissimilar, hosted, standalone and library, and self-compiled and third party executables. This covers the aforementioned requirements.

7.1.3 Empirical Selection of Similarity

In this section, we select a similarity function for the following evaluation. To do this, we investigate a random selection of alignments from the following evaluation.

We first investigate the F_1 scores achieved by the different similarity functions, sampled from all parts of the test suite (see Figure 7.1). The sampling is limited to 100 alignments per test suite from all possible alignments (all similarity functions present in equal amounts). The plot uses box plot notation from McGill, Tukey, and Larsen [26] (marker in box is median, left border of box is first quartile, right border of box is third quartile, whiskers range up to the largest value smaller than 1.5 times the distance between

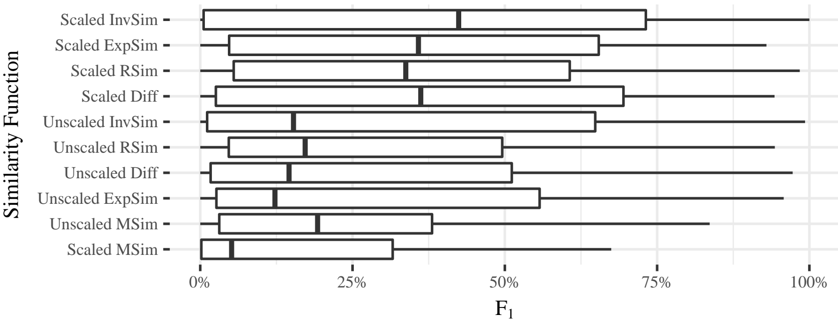


Figure 7.1: F_1 Scores of a Random Sampling of 100 Calculated Alignments per Test Suite Sorted by Mean

the first and third quartiles, and dots for outliers) and is sorted by the mean F_1 score of the different similarity functions.

The plot does not show that any similarity function clearly outperforms the other similarity functions. Additionally, regeneration (new sampling) of the plot leads to a different order of the similarity functions. But, the unscaled and scaled *maximum similarity* functions are usually positioned at the end of the list.

Since this simple statistical analysis does not yield a candidate similarity function to be used throughout the evaluation, we take a second approach and investigate the order (by mean) of the different similarity functions per test case.

Table 7.7 shows the order by mean of each similarity function per test case. The last column states the average position achieved by each similarity function. The top part of the table contains the similarity functions that are placed first at least once. The rows are sorted by average placement.

The general results show that most of the proposed similarity functions, except for the unscaled and scaled *maximum similarity*, can provide good alignments in several cases. We conclude that for the application scenario, quality control on the alignment result is more important than selecting an ideal alignment goal. Generally, selecting one of the similarity functions from the top part of the table seems preferable.

Table 7.7: Ranking of Mean Alignment Quality per Similarity Functions per Test Case

Similarity Function	glibc	redboot	musl	vmlinux	cpu2006	stamp	Bb.mtx.	∅
Unscaled Relative Similarity	1	1	6	2	3	2	1	2.3
Scaled Relative Similarity	7	2	1	1	10	4	9	4.9
Unscaled Exponential Similarity	8	9	10	8	1	1	5	6.0
Scaled Difference	3	3	2	3	6	7	4	4.0
Unscaled Difference	2	6	5	6	5	8	2	4.9
Unscaled Inverse Similarity	4	7	9	7	2	3	3	5.0
Scaled Exponential Similarity	5	4	4	5	8	5	8	5.6
Scaled Inverse Similarity	6	5	3	4	9	6	7	5.7
Unscaled Maximum Similarity	9	8	8	10	4	10	6	7.9
Scaled Maximum Similarity	10	10	7	9	7	9	10	8.9

To reduce the amount of discussion in the following case studies, we investigate only the alignments calculated by the raw relative similarity goal, as it is placed best on average. This similarity function seems to provide a good trade-off between exactness for identically sized functions and robustness against variations. In case of obvious flaws in the result, we additionally investigate results from the other similarity functions.

7.1.4 Canonical Case Study

In this section, we discuss the canonical layout and content of the case studies in the following evaluation. Each case study begins with a summary table that provides a rough overview of the data. The individual aspects of each case, namely the executable similarity, the sequence stability, the alignment quality, the quality estimate, and the effectiveness, are discussed in separate paragraphs.

The summary table starts with a summary of the test cases properties. The data for the similarity, stability, and quality is summarised using a five-number summary with the addition of a mean, and with a range indicator. Here, we use a block in a box, where the box covers the full range from 0 to

100%, and the left and right borders of the block mark the first and third quartile, respectively. For example, ▭▮▭ represents a first quartile of 50% and a third quartile of 66%.

The estimate and effectiveness data is shown using Spearman's ρ, a custom prediction scheme (described later in this section, together with the second style of plots), and a two bar summary indicator. Spearman's ρ (see [39]) is a rank correlation coefficient. Such a rank correlation coefficient assesses the possibility to express the relation of the elements using a monotone function. The ρ part of the result is positive for an increasing monotone relation, and negative for a decreasing relation. ρ approaches zero for uncorrelated data, and one for a perfect rank correlation. The p-value gives a probability that the values are actually uncorrelated, and can thereby be used to inspect the amount of information contained in ρ. The two bar indicator counts the number of outliers for the custom prediction scheme. The black bar stands for the 67% prediction, the grey bar stands for the 75% prediction. The height of the bar describes the ratio of outliers to the number of alignments for which the prediction holds. A full height bar means no outliers. For example, ▄▆ would describe 50 outliers from 100 elements for the 67% prediction, and 25 outliers from 100 elements for the 75% prediction.

The assessment of similarity, stability, and quality uses the same style of plots. To be easily identifiable, all similarity plots use orange colours, all stability plots use green colours, and all quality plots use blue colours. The plots use a shared scale with five equally wide intervals, to allow a comparison of the plots beyond a single case. More details on the values can be read from the summary table. The data is grouped into facets by their shared attributes (e.g., a shared optimisation level). Each plot only contains elements in the upper-right triangle, since the additional positions in the plot are symmetrical to the diagonal, and the elements on the diagonal are identical.

Each of the inspections follows the same pattern. First, the numerical range of the elements is inspected. Secondly, the plot is inspected for visible patterns in the values. Lastly, an overall assessment on the values is made.

For the executable similarity, the plots show the *similar* value described in Equation (7.1), which relates the number of identically named functions to the number of unique names for each executable pair. This allows an

inspection of the effect of the variations of each case on the ground truth similarity of the executables.
For the sequence stability, the plots show the *recall* values of alignments calculated using a similarity function on the function names as described in Equation (7.2). This allows an inspection of the ground truth information about the amount of functions that are identically named and appear in the same order in an executable pair.
For the alignment quality, the plots show the F_1 score (see Equation (7.4)) of alignments calculated using a function-size based similarity function. This allows an inspection of the quality of the alignments that the function size similarities achieved using ground truth

We use a second style of plots for inspecting quality estimates and the effectiveness of the alignment algorithm. These are scatter plots of the F_1 scores against the quality estimate (the *alignment similarity*) or the similarity (*similar*), respectively. The variations in the data (identity or difference in certain attributes) are shown using different shapes and colours. The value areas relevant to the custom prediction scheme of alignment quality are annotated as rectangles with a solid (67%) or dashed (75%) outline.

Each of the inspections follows the same pattern. First, the plot is inspected for visible patterns in the correlation. Secondly, the custom prediction scheme is evaluated. Lastly, Spearman's ρ of the correlation is inspected.

In the discussion of the test cases, care has to be taken in the interpretation of this correlation data, as the parameters of the test cases sometimes imply the coverage of only parts of the scale (e.g., dissimilar executables have a low similarity). In Section 7.3, we summarise the test cases into one plot and draw conclusions about the correlation on the whole.

To simplify the interpretation of quality estimates and effectiveness, we define two areas in the plot, which are blank for ideal measures, and may contain elements in case of suboptimal quality estimates or effectiveness. This allows simple "warning light" conclusions about the results. These areas are equivalent to a simple implication:

$$\text{measure} > v \Rightarrow F_1 > v \tag{7.5}$$

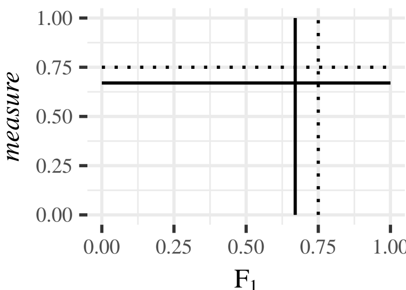


Figure 7.2: Construction of Interpretation Areas in Plot

Figure 7.2 shows a sketch of the implication for v being 67% (solid line) and 75% (dashed line). For example, for the solid line the implication says that any point above the horizontal line also needs to be on the right of the vertical line. Points below the horizontal are not of concern for the implication, and points above the horizontal and left of the vertical are counterexamples for the implication. In later plots we only annotate these counterexamples. We use these implications for both the quality estimate and the similarity measure.

For the inspection of the quality estimate, an ideal quality estimate would result in a plot with a monotonic function or even better, in a diagonal line, which would describe a quality estimate that describes the true quality without any error or uncertainty. If elements of the quality estimate plots leak into the above defined areas, this hints to a quality estimate not usable for our purpose. In the detailed inspection, we show all three proposed estimates, namely match similarity (Equation (6.1)), score ratio (Equation (6.2)), and alignment similarity (Equation (6.3)), but in the summary table, we only show the generally best performing *alignment similarity*.

For the inspection of the effectiveness, if the assumption on the sequence stability holds, a well performing alignment technique is expected to produce a plot that resembles a monotonic function. This would indicate that with growing similarity in the executable pairs the technique recovers more data. If the plot shows outliers above a diagonal structure, this indicates that

these outliers have a high similarity compared to the achieved F_1 score and the technique was not effective enough compared with this similarity. For outliers below, this indicates that the techniques returned better results than expected from the executable similarity. To simplify this analysis, we use an annotation in the plot that separates the plot into three sections: above, on, and below the diagonal. Additionally, we use the same implication as for the quality estimate, to highlight areas in which outliers clearly indicate a suboptimal effectiveness.

7.2 Evaluation Data

In this section, we discuss the evaluation results for the sets of similar, dissimilar, and validation executables. Each case is described in the canonical style defined in the last section.

7.2.1 Product Lines

The original definition of product lines in this work lists changing revisions, changes to the build process, changes in feature configuration, and adaptions to the platform as varying factors for product lines. In this section, we use four examples to cover the product line variations and to exemplarily assess how well the proposed technique copes with the encountered variations.

`glibc`							
Content	C Library						
Variation	Versions: 5, Optimisations: 4						
	Min.	1Q.	Med.	Mean	3Q.	Max	
Similarity	73	76	78	81	82	98	
Stability	88	92	96	95	97	100	
Quality	76	84	90	89	94	100	
		ρ	p-value		>0.67	>0.75	
F_1 ˜ Aln. Similarity		0.13	0.1		153/0	153/0	
F_1 ˜ Similarity		0.79	$2 * 10^{-16}$		153/0	153/0	

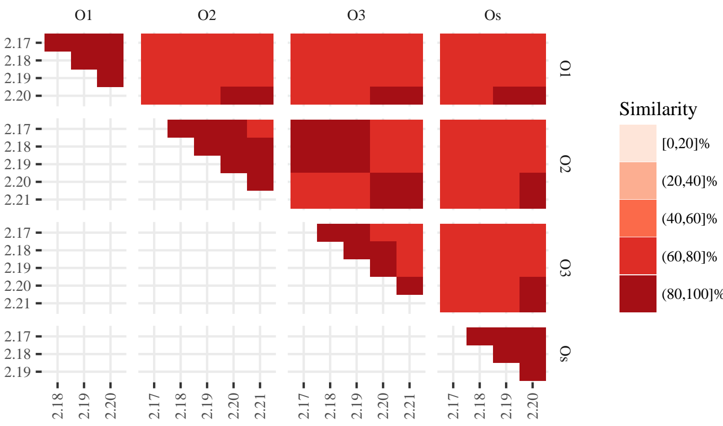


Figure 7.3: Similarity of `glibc` Library Pairs

The `glibc` library is a C standard library. It is compiled using gcc, with varying optimisation levels, for a small set of versions. Not all versions are present for all optimisation levels, since they do not compile with the version of gcc installed on the evaluation machine.

Similarity The similarity range shows that at least 73% of the function names are shared between executables pairs.

Figure 7.3 shows a plot of the similarities of the pairs of `glibc` libraries. The elements represent single versions of the library, which are grouped by the optimisation levels. The highest ratios of similarities are within identical optimisation levels and for single-step increases of optimisation levels. Overall the `glibc` library pairs are highly similar for all variations.

Sequence Stability The recall ranges between 88 and 100%. This means that at least 88% of the identically named functions between any two `glibc` library samples in the test suite were preserved in sequence.

Figure 7.4 shows the recall for the alignments calculated with name identity comparisons. Since all plots share a common scale, the plot only shows a single class of high sequence stability. Although this hides details

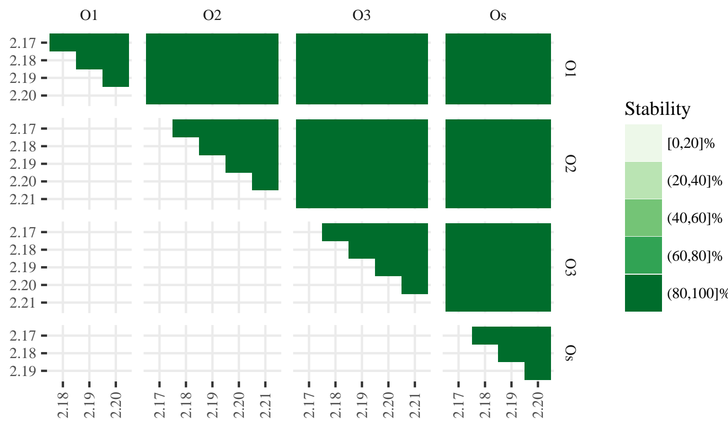


Figure 7.4: Stability (Recall of Name Identity Alignment) of `glibc` Library Pairs

which pairs have the most stable sequence, the sequence stability assumption holds for these executables.

Size Alignment Quality The F_1 scores of the raw relative similarity alignments range between 76% and 100%. This means that in the best case, all of the identical functions are among those returned by the alignment (recall part) and that in this case, only the identical functions are returned by the alignment (precision part).

Figure 7.5 shows the alignment quality using the F_1 score for alignments with *raw relative similarity* comparisons. The results show that only in some cases, for certain variations in optimisation levels, the F_1 score did not reach very high levels, but overall, the F_1 score is in the high category. The best results for reconstruction are the cases where the library executables were of the same optimisation level or of the same version. In most of these cases with identical optimisation levels, the F_1 scores are above 95% (not shown).

Quality Estimate The correlation of *alignment similarity* to the F_1 score using Spearman's ρ shows a low p-value of 0.1, and a ρ of 0.13. This

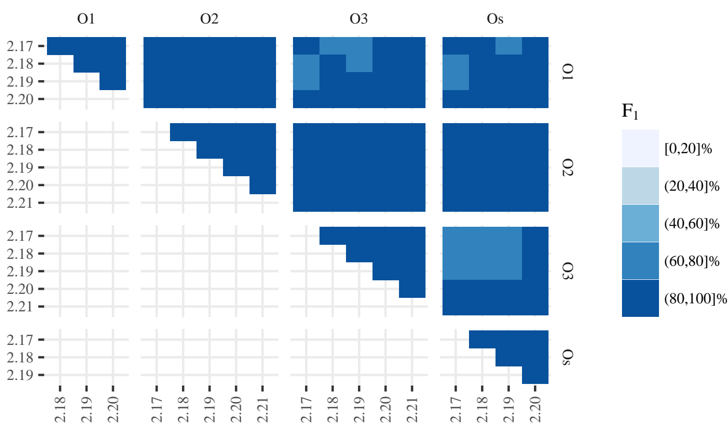


Figure 7.5: Quality (F_1 Scores) of Alignments of `glibc` Library Pairs using Unscaled Relative Similarity

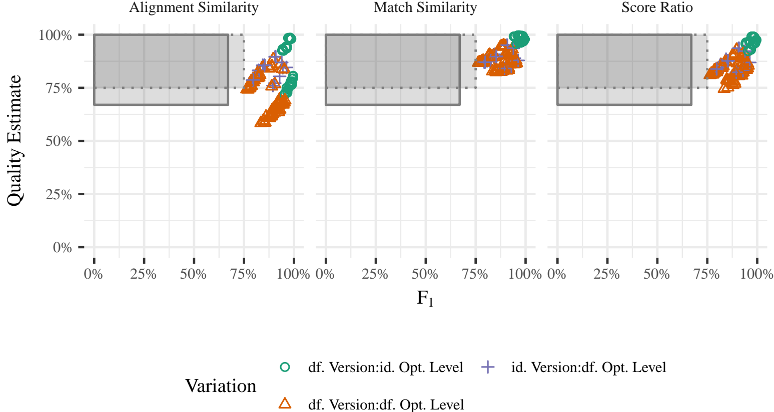


Figure 7.6: Correlation of Quality Estimates with F_1 Scores of `glibc` Library Pairs

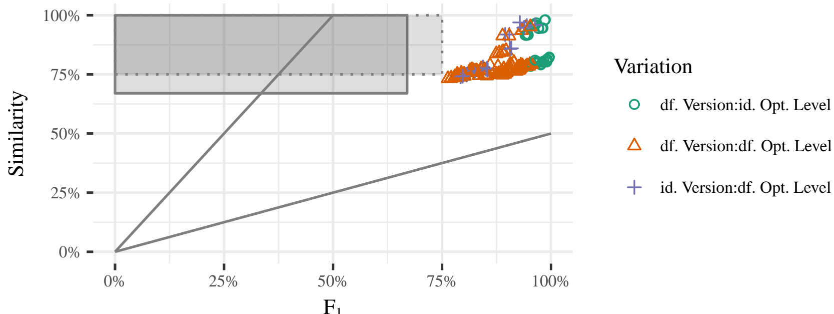


Figure 7.7: Correlation of Similarities with F_1 Scores of `glibc` Library Pairs

indicates that there is no clear numerical correlation. Figure 7.6 shows the correlations of F_1 scores with the *alignment similarity* of the executable pairs. The plot of the *alignment similarity* shows a weak correlation with the F_1 scores on the full range. Both implications hold trivially for the data, as all of the *alignment similarities* as well as the F_1 scores are above 67%. The *match similarity* and *score ratio* behave similar to the *alignment similarity* in this case.

Effectiveness The correlation of F_1 scores with the similarity (see Figure 7.7) shows that for higher similarities, the alignment has a tendency to return better alignments (ρ: 0.79). There is no group of outliers above the diagonal where the similarity exceeds the F_1 score. Both implications trivially hold for the data, as all of the similarities as well as the F_1 scores are above 0.67.

This case study shows that the unscaled relative similarity can cope with executables of high similarity, for which the stability assumption holds, and achieve high quality in the alignments.

redboot							
Content	Embedded Bootloader						
Variation	*Platform*, Architecture, (unknown: Version, Compiler, Optimisation)						
	Min.	1Q.	Med.	Mean	3Q.	Max	
Similarity	23	40	46	48	56	100	
Stability	66	79	84	85	91	100	
Quality	5.6	34	50	50	62	100	
		ρ	p-value	>0.67	>0.75		
F_1 ~ Aln. Similarity		0.58	$2 * 10^{-16}$	378/0	378/0		
F_1 ~ Similarity		0.68	$2 * 10^{-16}$	374/4	378/0		

The `redboot` examples potentially cover all of the order-preserving variations. The executables are obtained from the example platforms section of the `redboot` website, where they are not tagged with a specific revision, build process or feature configuration. This closely represents the expected use case of the differential analysis (DA) when analysing software from embedded systems.

We suspect the executables to be built by a small set of persons using similar or identical build processes. The revision level and set of enabled features is suspected to change significantly as different features are needed for the different platforms within the architectures, and the executables are probably not rebuilt regularly, but only when significant changes for a platform are made.

Similarity The value range of the similarity is large, but for most of the executable pairs, about half of the function names are shared (first quartile 40%, median 46%, third quartile 56%). Figure 7.8 shows a plot of similarities between the `redboot` executables. The pairs are grouped by their architecture and the individual elements stand for different platforms. Several of the elements seem to form groups of higher similarity, but in general, the similarities do not contain a pattern. Overall, most of the executables are of medium similarity.

Sequence Stability The recall ranges between 66% and 100%. This means that in the worst case, 66% of the identical names between two executables

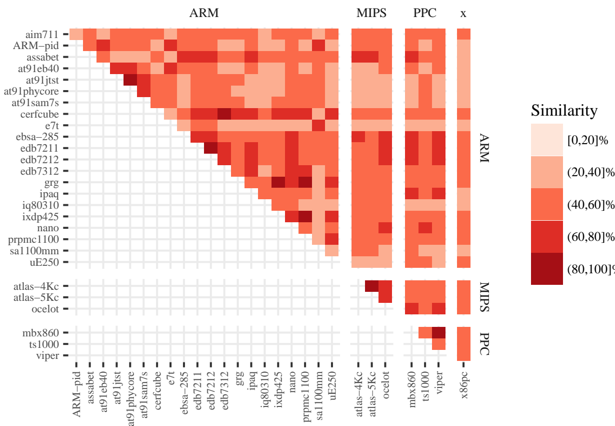


Figure 7.8: Similarity of `redboot` Executable Pairs

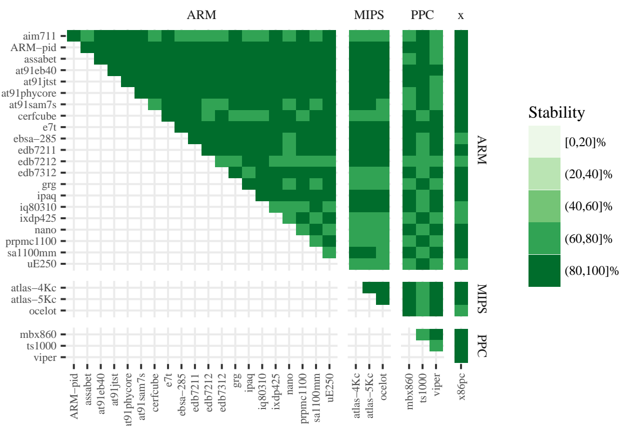


Figure 7.9: Stability (Recall of Name Identity Aligment) of `redboot` Executable Pairs

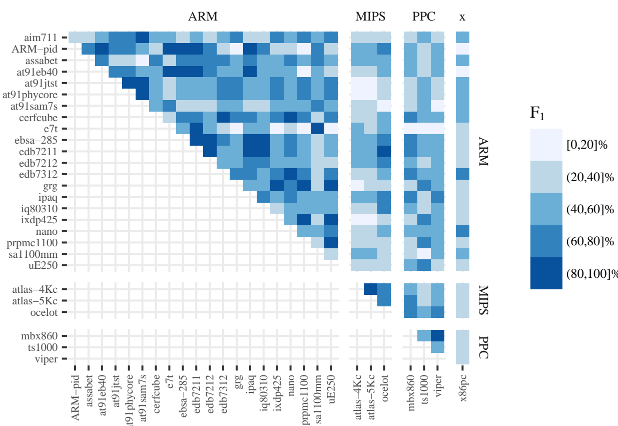


Figure 7.10: Quality (F_1 Scores) of Alignments of `redboot` Executable Pairs using Unscaled Relative Similarity

are in sequence. Figure 7.9 shows the recall of the alignments calculated using name identity. The sequence stability seems to be generally higher than the executable pair similarity. Overall, the stability of the sequence is high.

Alignment Quality The quality of the alignments ranges from very low (5.6%) to perfect (100%). Figure 7.10 shows the F_1 scores of the alignments between the `redboot` executables using unscaled function lengths and relative similarity. The plot shows a slight increase in scores for executable pairs within the same architecture. For all facets of differing architectures, the scores have a maximum of about 80%. But still, even for the different architectures the median score of the alignments is above 24.5%. This behaviour is expected, as for differing architectures, function lengths are not expected to be identical, as the number of instructions used to implement a function as well as the length of individual instructions changes significantly. Overall, the quality of the alignments is medium, but with a high variance, reaching into low as well as high classifications.

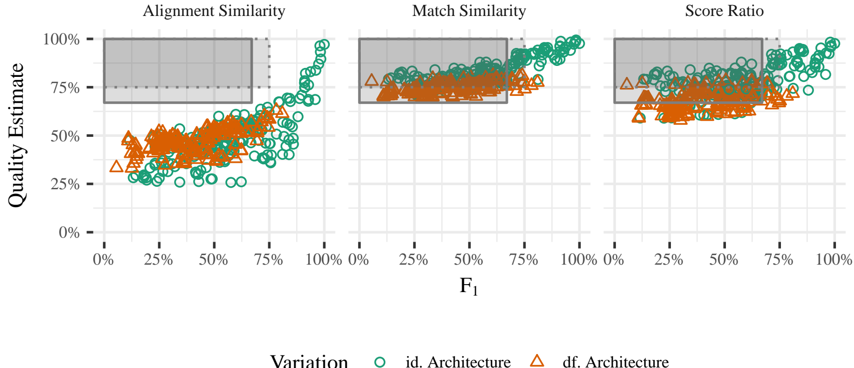


Figure 7.11: Correlation of Quality Estimates with F_1 Scores of `redboot` Executable Pairs

Quality Estimate Figure 7.11 shows the correlation plots between the F_1 score and the proposed quality estimates. The plots show a correlation between the *alignment similarity* and the F_1 score. Additionally, the alignment similarity shows a steep increase in F_1 scores for *alignment similarities* better than 70%. This is a set of alignments where the *alignment similarity* is a useful indicator for the achieved alignment quality, since there are alignments of all qualities present. The implications define above hold, which means that for high *alignment similarity*, the F_1 is also always high. In contrast to this, both the *match similarity*, and the *score ratio* begin to show an increase in their quality estimates in cases where the actual quality is low (F_1 score less than 25%). In Section 7.3, when we summarise all cases, this behaviour is visible even more. Numerically, Spearman's ρ of the *alignment similarity* against the F_1 scores is 0.58 with a highly significant p-value, confirming the visual correlation.

Effectiveness Figure 7.12 shows a correlation plot between the F_1 score and the similarity. It shows a correlation, with no group of outliers above or below the diagonal area. Outliers above would indicate cases where the achieved quality did not reach the potential from the similarity. Outliers

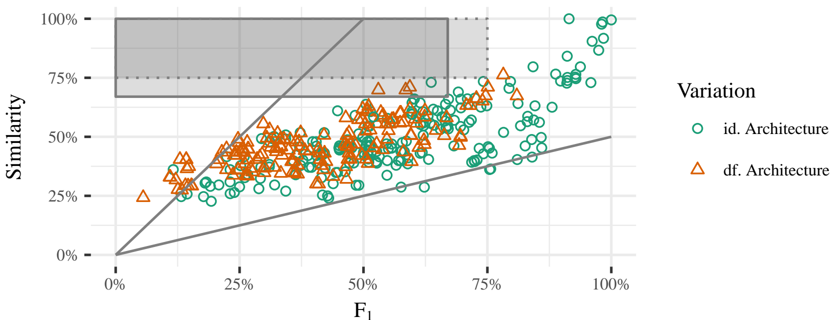


Figure 7.12: Correlation of Similarities with F_1 Scores of `redboot` Executable Pairs

below would indicate cases where the alignment achieved better quality than expected from the similarity. For different architectures, the *similarity* as well as the F_1 score reach a maximum of about 80%. This leads to the expectation that for executables of different architecture with higher similarity, higher F_1 scores could also be expected. The custom prediction scheme is valid for 75% values, for 67% values, a small number of outliers exist. Numerically, Spearman's ρ of the *similarity* against the F_1 scores is 0.68 with a highly significant p-value, confirming again the visual correlation.

This case study shows that the unscaled relative similarity can cope with executables of medium similarity, for which the stability assumption holds, and achieve medium quality in the alignments. Both implications of the prediction scheme hold (with 4 outliers), showing that the quality estimate using *alignment similarity* is usable, and no executables of high similarity have alignments of medium or low quality.

musl							
Content	C Library						
Variation	Version, *Compiler*, Optimisation						
	Min.	1Q.	Med.	Mean	3Q.	Max	
Similarity	50	65	73	76	88	100	
Stability	69	90	93	92	96	100	
Quality	8.8	23	46	53	84	100	
	ρ	p-value	>0.67	>0.75			
F_1 ~Aln. Similarity	0.95	$2 * 10^{-16}$	2275/3	2278/0			
F_1 ~Similarity	0.87	$2 * 10^{-16}$	1616/663	2063/215			

The `musl` library is a C standard library. It is compiled using gcc or clang, with varying optimisation levels, for a large set of versions. Originally, the set contained all release steps between 0.9.0 to 1.1.10, as well as the optimisation levels -O0, -Os, -O1, -O2, -O3. To reduce the amount of alignments to calculate and plot, the dataset was reduced to only contain the even versions and the optimisations -O0 and -O3.

Note that the version step from 0.9.4 to 0.9.6 introduced 500 additional function in the -O0 versions of the library. For the higher optimisation level, these additional functions seem to be completely inlined.

Similarity The similarities of the executables range from 50% to 100%, with a median of 73% and a mean of 76%. The first quantile is 65% and the third quantile is 88%. Figure 7.13 shows a plot of the similarities between the `musl` library pairs. The plot indicates the change between the version 0.9.4 and 0.9.6, since the magnitude of the similarities shows a step for most pairs before and after these versions. This step increase in similarities seems to be suppressed by the use of the -O3 compiler flags, as it is not present for pairs of executables that share this optimisation level.

Sequence Stability Figure 7.14 shows the recall of the alignments calculated using name identity. The alignments are grouped by their architecture. The recall ranges between 69% and 100%. Again, the data shows a notable change in recall levels of alignments between executables before and after the 0.9.4 to 0.9.6 version change (in the plot, due to the global scale, only visible for gcc, -O0 vs. clang, -O0). In contrast to the change in similarities

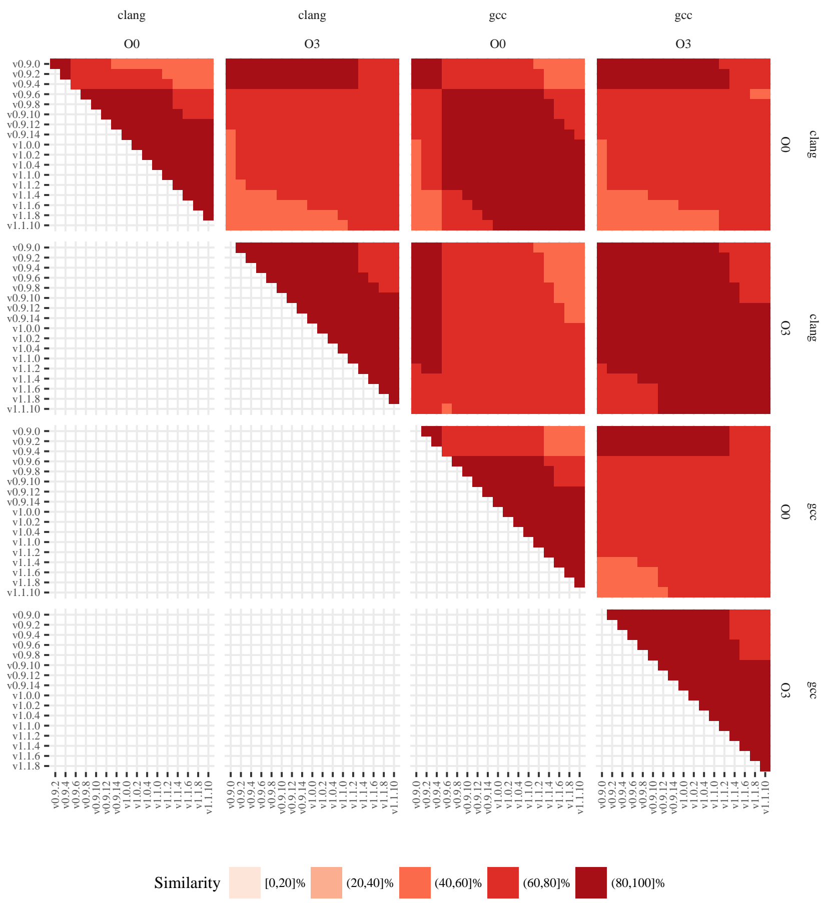


Figure 7.13: Similarity of `musl` Library Pairs

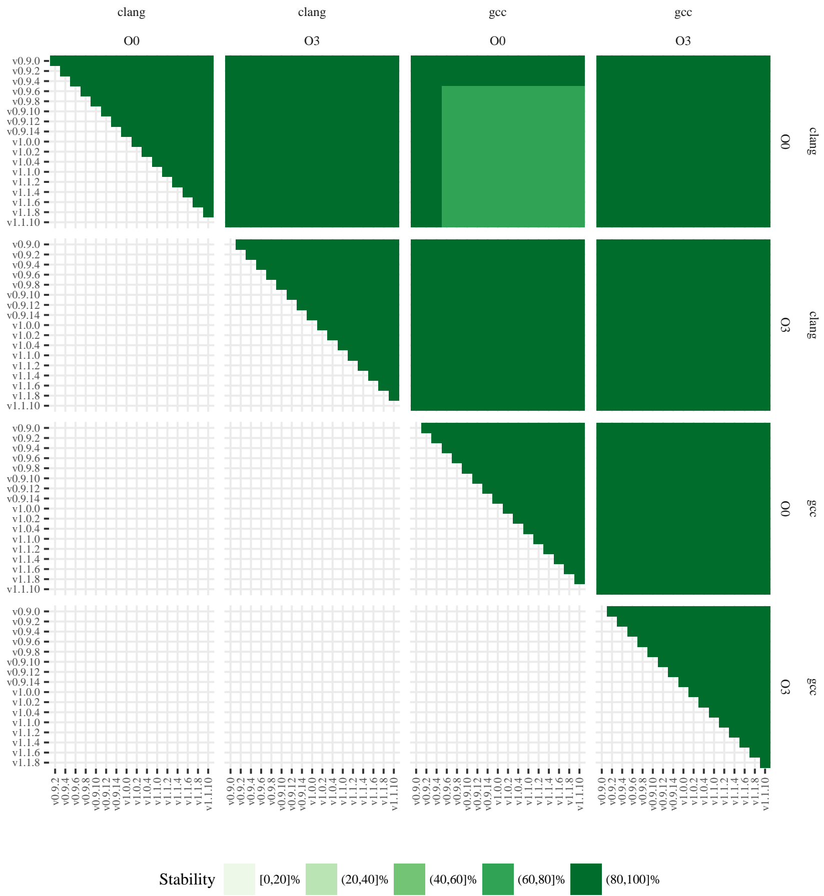


Figure 7.14: Stability (Recall of Name Identity Alignment) of `musl` Executable Pairs

discussed above, this change is present for alignments with a shared -O3 optimisation level.

Alignment Quality The quality of size similarity alignments is discussed for the best overall average optimisation goal (raw relative similarity) and the best optimisation goal for this case study (scaled relative similarity).

Figure 7.16 shows a plot of the F_1 scores for alignments calculated using scaled size similarities, and Figure 7.15 shows a plot of the F_1 scores for alignments calculated using raw size similarities. Both plots show the same invariance to the version step between 0.9.4 to 0.9.6 as the similarities between the executables for the shared optimisation level -O3.

The shapes in the plot are similar, with the raw function length slightly outperforming the scaled function length with 40% of the alignments being better than 72% compared to 63% for the latter. But the scaled function lengths are able to reconstruct better identity information for combinations of compiler and optimisation level, where these attributes differ.

Quality Estimate Figure 7.17 shows the correlation plots between the F_1 score and the proposed quality estimates. The plots show a correlation between the *alignment similarity* and the F_1 score. The implications defined above hold (with three outliers) for the *alignment similarity*, which means that for high *alignment similarity*, F_1 is also always high. Both the *match similarity* as well as the *score ratio* show a flatter correlation, but without a clear non-monotone behaviour. In this case, these measures could also be used as a quality estimate. Numerically, Spearman's ρ of the *alignment similarity* against the F_1 scores is 0.95 with a highly significant p-value, confirming the visual correlation.

Effectiveness Figure 7.18 shows a correlation plot between the F_1 score and the similarity. It shows several groups of outliers. Although these groups are nicely separated from the cases of high similarity and high F_1 scores, these groups cannot be identified by the variations annotated in the plot. A probable scenario is that these cases are associated with the change introduced between 0.9.4 and 0.9.6. Figure 7.19, which shows the same correlation of similarity and F_1 score, but for the alignments calculated with scaled relative similarity, confirms this, as there are nearly no outliers outside of the diagonal section of the plot. In neither of the plots, the prediction scheme is valid. This indicates that neither of the inspected similarity functions can exploit the full potential in the executable

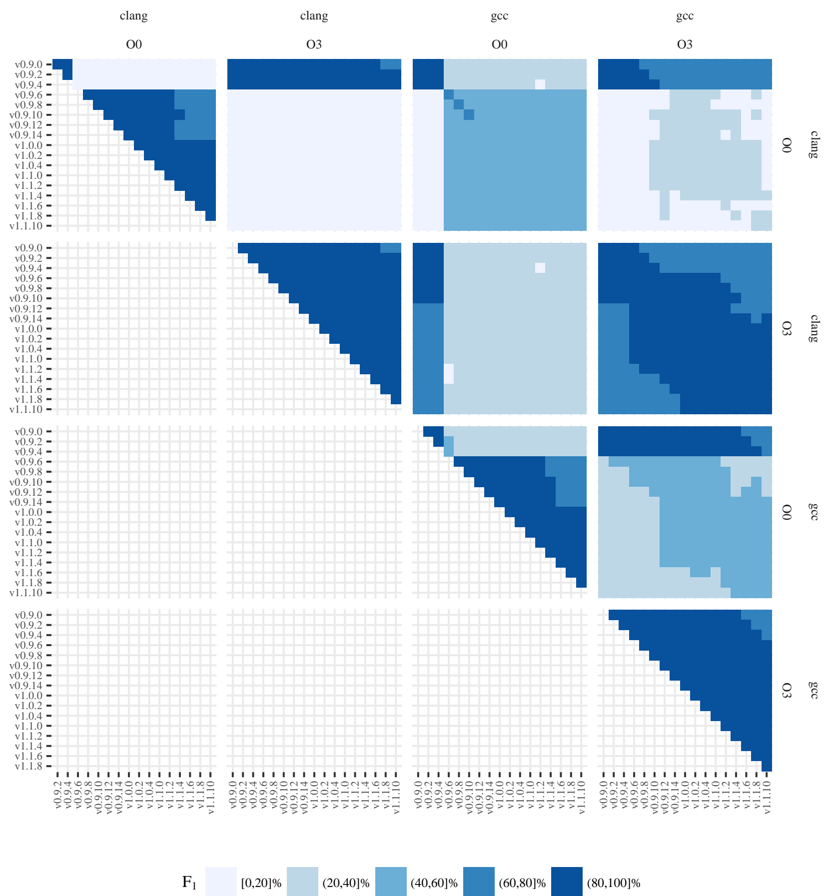


Figure 7.15: Quality (F_1 Scores) of Alignments of `musl` Library Pairs using Unscaled Relative Similarity

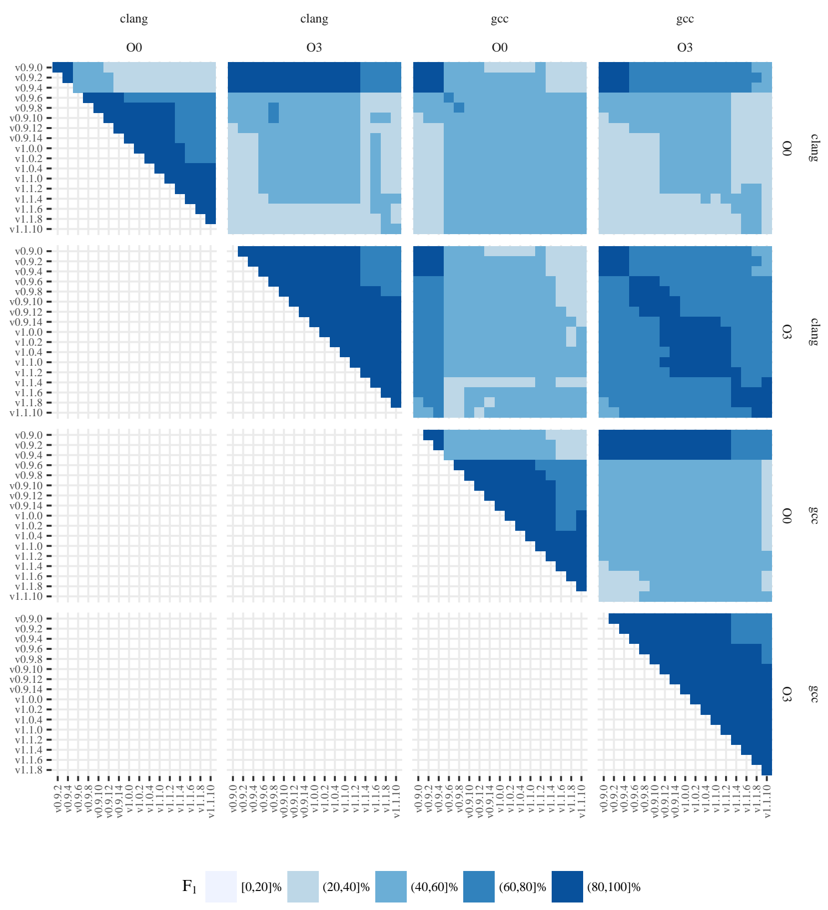


Figure 7.16: Quality (F_1 Scores) of Alignments of `musl` Library Pairs using Scaled Relative Similarity

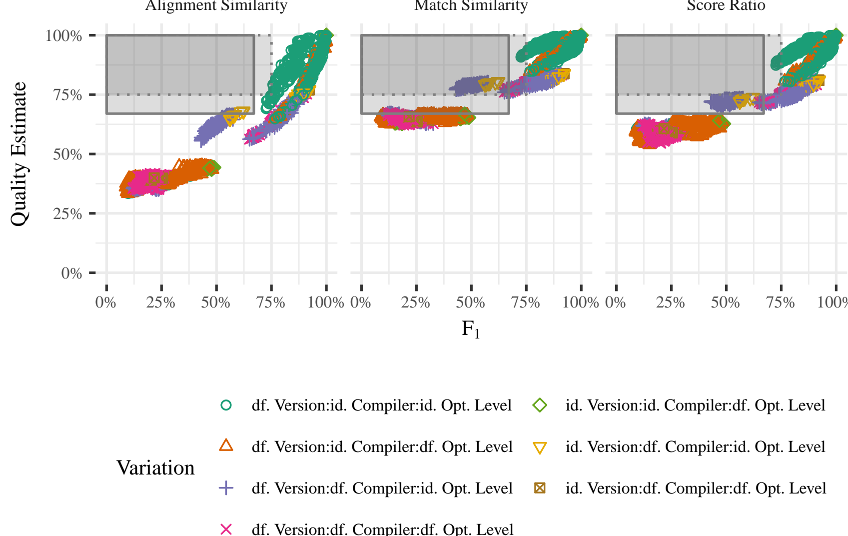


Figure 7.17: Correlation of Quality Estimates with F_1 Scores of `musl` Library Pairs

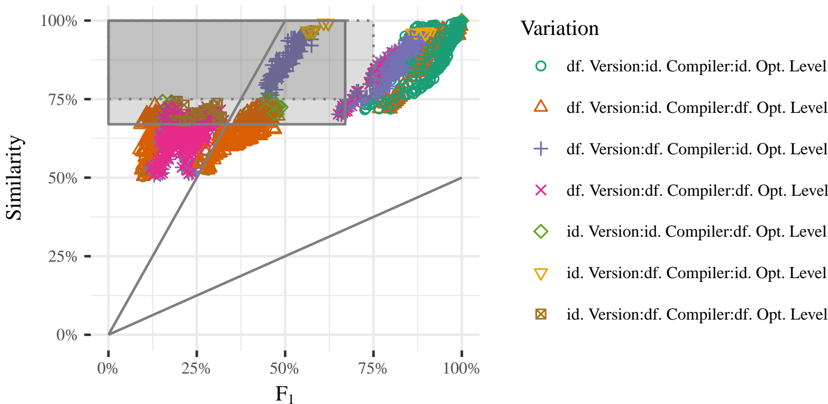


Figure 7.18: Correlation of Similarities with F_1 Scores of `musl` Library Pairs for Alignments using Unscaled Relative Similarity

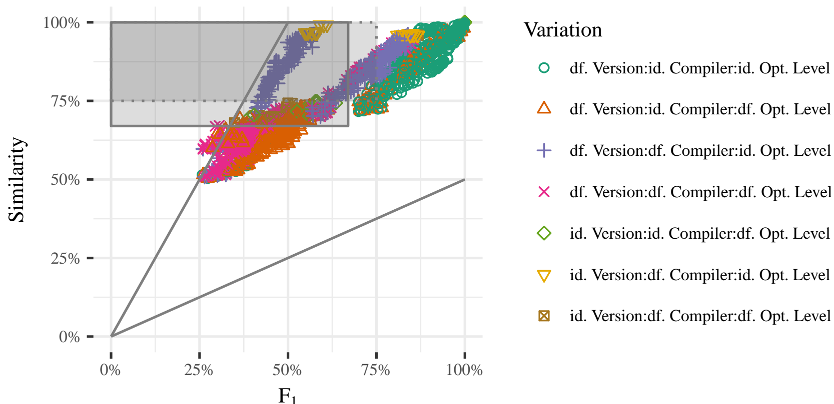


Figure 7.19: Correlation of Similarities with F_1 Scores of `musl` Library Pairs for Alignments using Scaled Relative Similarity

similarities. Numerically, Spearman's ρ of the *similarity* against the F_1 scores is 0.87 with a highly significant p-value, which confirms a general correlation of the data.

This case study shows that, although the unscaled relative similarity cannot develop to the full potential, the scaled relative similarity can improve the results. Only the implications for the quality prediction using *alignment similarity* hold, which shows that although alignments of low quality exist, the quality estimate can warn about these.

vmlinux							
Content	Operating System Kernel						
Variation	*Architecture*, Version						
	Min.	1Q.	Med.	Mean	3Q.	Max	
Similarity	20	27	59	53	77	97	
Stability	53	65	71	74	81	99	
Quality	0	5.2	34	37	62	98	
	ρ	p-value	>0.67		>0.75		
F_1 ˜Aln. Similarity	0.93	$2 * 10^{-16}$	1953/0		1953/0		
F_1 ˜Similarity	0.94	$2 * 10^{-16}$	1588/365		1717/236		

The `vmlinux` executables are uncompressed executables of the Linux kernel that can be executed directly and analysed without the need for unpacking. The test suite consists of versions 3.0 to 4.0 in 0.1 steps, compiled using gcc with default configurations for the AMD64, X86, and ARM32 architectures.

Similarity The similarity spread of the executables is large (from 20% to 97%), since the ARM32 executables have low similarity compared to the other architectures, but since the similarity is not compensated for executable sizes, this is expected, as the ARM32 executables only consist of about one third the number of functions that the other architectures have. Figure 7.20 shows the plot of similarities, which indicates that executable pairs with the same architecture generally have a medium to high similarity.

Sequence Stability The sequence stability is medium to high (min 53%, median 71%, max 99%). Figure 7.21 shows the plot of recall of the alignments

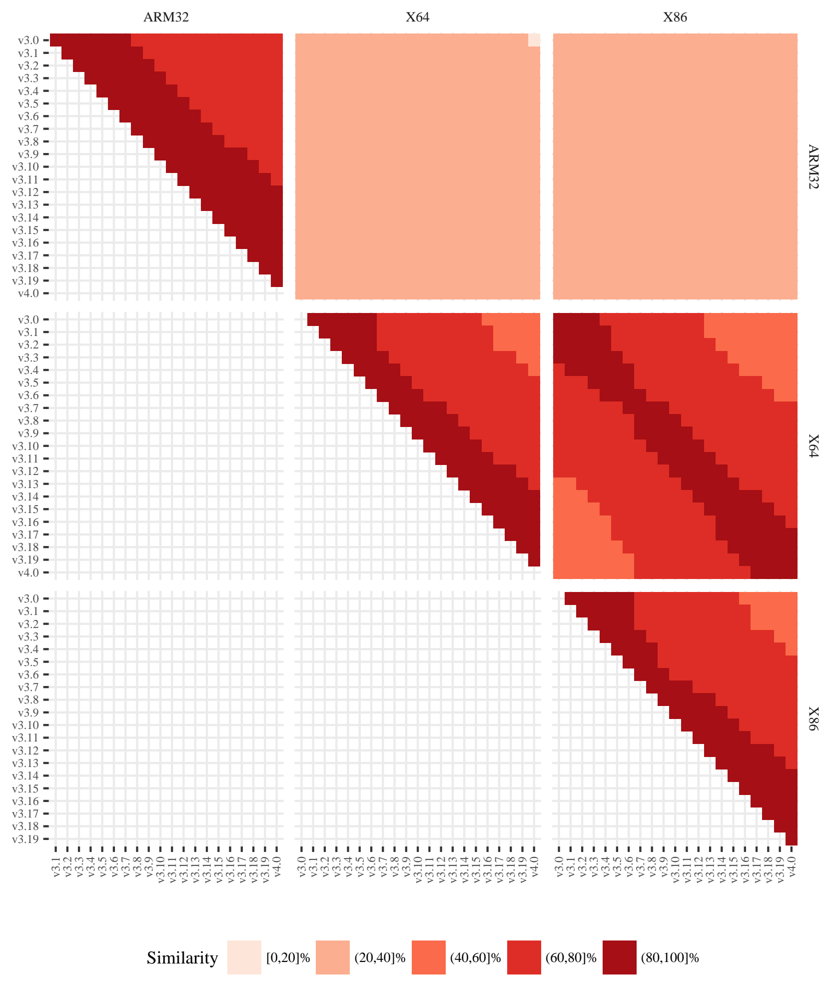


Figure 7.20: Similarity of `vmlinux` Executable Pairs

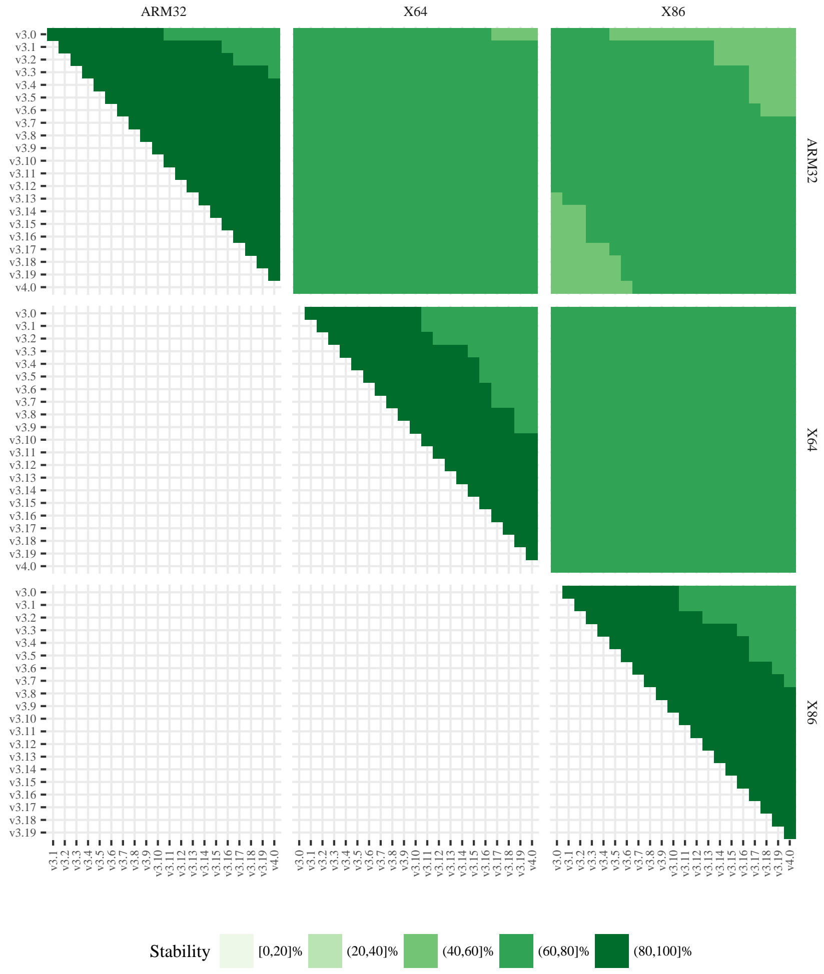


Figure 7.21: Stability (Recall of Name Identity Alignment) of `vmlinux` Executable Pairs

using function name identity, which indicates that for executable pairs with the same architecture, the stability is high, especially for pairs with a small difference in version number. But, corresponding to the general value range, even for the executable pairs of differing architecture, at least half of the common functions are in sequence.

Alignment Quality The range of alignment quality for unscaled relative similarity alignments is nearly maximal (min 0%, max 98%), but, since the third quartile is at 64%, is categorised as low to medium. Figure 7.22 shows the plot of the unscaled relative similarity alignments. It shows, corresponding with the similarity and stability data, that for the executable pairs with the same architecture and a small difference in version, the alignments are of high quality. Nonetheless, the alignments between ARM32 and the other architectures are of very low quality. Only the alignments between X86 and X64 are of low to medium quality.

Quality Estimate Figure 7.23 shows the correlations between the *match similarity*, the *score ratio*, and the *alignment similarity* with the F_1 scores for the `vmlinux` executable pairs. The plot annotates the differences in the executables using shapes and colours. The plot shows a separated group of low quality alignment, consisting of executable pairs with different architecture. These are the alignments with the ARM32 architecture. For this group, the *match similarity* and *score ratio* show significantly too high quality estimates. Only the *alignment similarity* shows a nice correlation. Spearman's ρ is 0.93 with a highly significant p-value, which confirms this correlation.

Effectiveness Figure 7.24 shows a correlation plot between the F_1 score and the similarity. It shows a correlation, with one group of outliers. It is the same group as above, with the inter-architectural alignments with the ARM32 architecture. These alignments should have performed better from a similarity point of view. Concerning the prediction scheme, both implication areas contain a large number of counterexamples (365 and 236). Spearman's ρ is 0.94 with a highly significant p-value, which at least confirms a good tendency of the alignments to be of better quality with increasing similarity.

The case study shows that within the architectures, the alignment can produce results of very high quality. It is even possible to achieve medium quality results between architectures from the same heritage (X86 and X64).

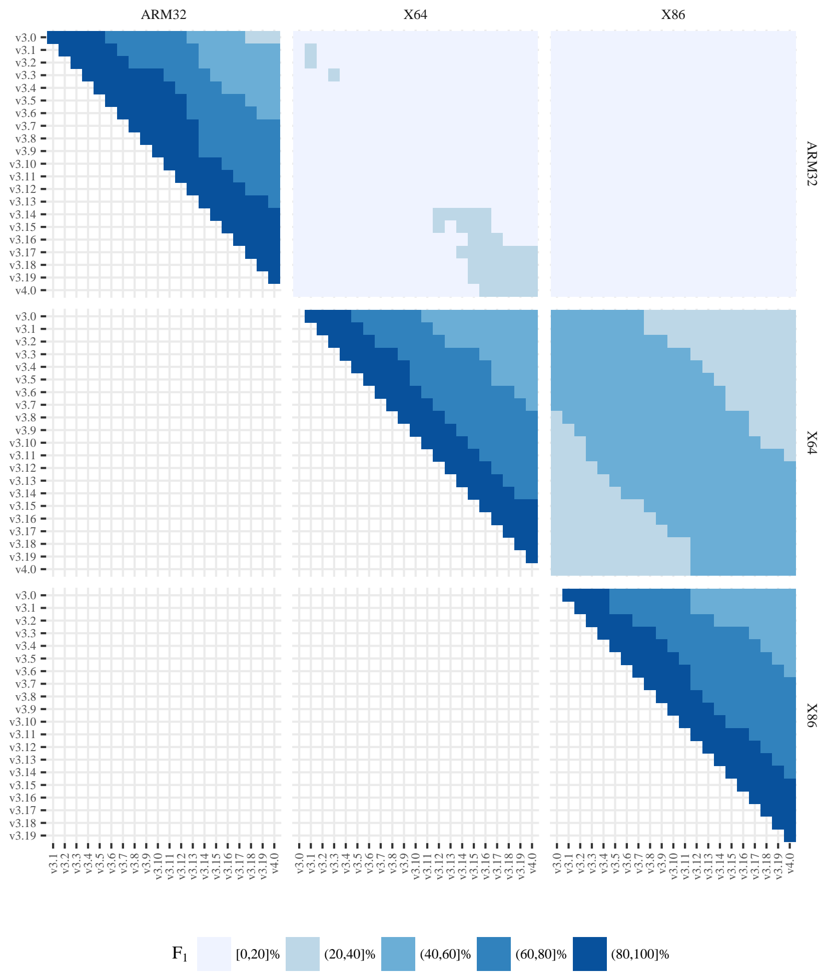


Figure 7.22: Quality (F_1 Scores) of Alignments of `vmlinux` Executable Pairs using Unscaled Relative Similarity

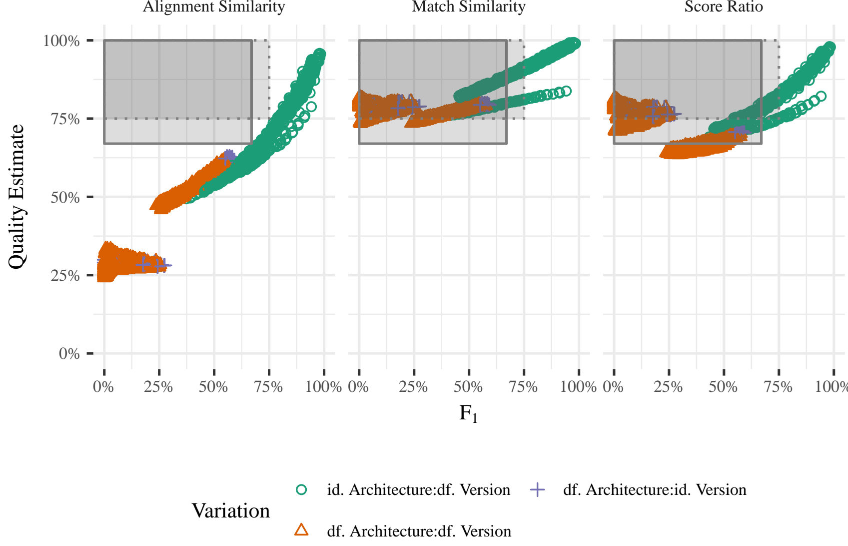


Figure 7.23: Correlation of Quality Estimates with F_1 Scores of `vmlinux` Executable Pairs

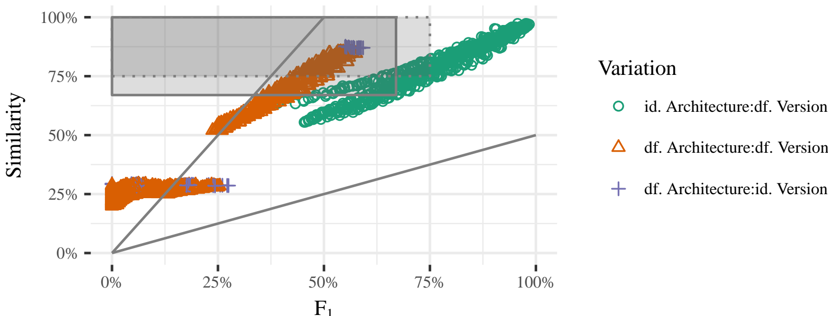


Figure 7.24: Correlation of Similarities with F_1 Scores of `vmlinux` Executable Pairs

7.2.2 Dissimilar Executables

In this section, we look at a test suite of dissimilar executables.

CPU2006							
Content	CPU Performance Evaluation						
Variation	Element						
	Min.	1Q.	Med.	Mean	3Q.	Max	
Similarity	0.16	1.4	3	5.7	8.6	33	
Stability	24	51	76	74	100	100	
Quality	0.19	3.6	8.3	11	14	59	
		ρ	p-value	>0.67	>0.75		
F_1 ~ Aln. Similarity		-0.15	0.03	189/1	189/1		
F_1 ~ Similarity		0.39	$4 * 10^{-8}$	190/0	190/0		

The CPU2006 executables do not cover the variations that are investigated in the last section, but simply are different executables that happen to be part of the CPU2006 benchmark.

The 19 executables for this test originate from a local compilation of the C and C++ subset of the SPEC CPU2006 benchmark. It includes source code for very different programs, all of which are compiled using gcc as 32 bit ELF executables for linux.

Executable Similarity The similarities range from 0.16% to 33%. As expected, these similarities for the different executables are low. The higher similarities are reached for pairs of executables (see Figure 7.25) where both executables have a small number of functions.

Sequence Stability The stabilities (recall of function name identity alignment) range from 24% to 100% with a median of 76%. These can be classified as medium to high stabilities. These high stabilities come partially as a surprise, since the executables are supposed to be dissimilar, and although they contain functions with identical names, it is surprising that these should appear in the same order. But, the stable parts of the function sequence mostly stem from the wrapper functions belonging to the dynamically linked C-library, and are therefore present at the same

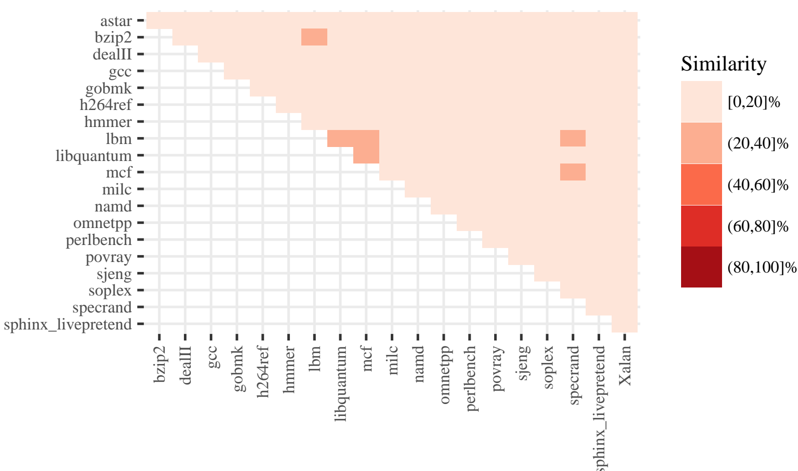


Figure 7.25: Similarity of `CPU2006` Executable Pairs

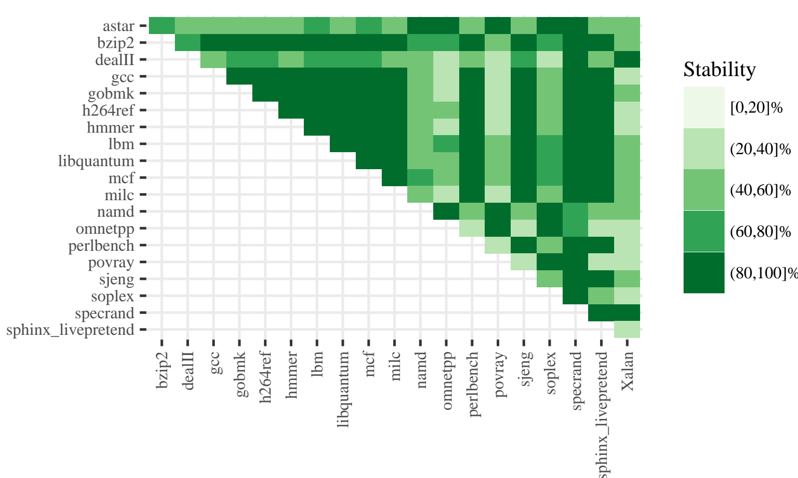


Figure 7.26: Stability (Recall of Name Identity Alignment) of `CPU2006` Executable Pairs

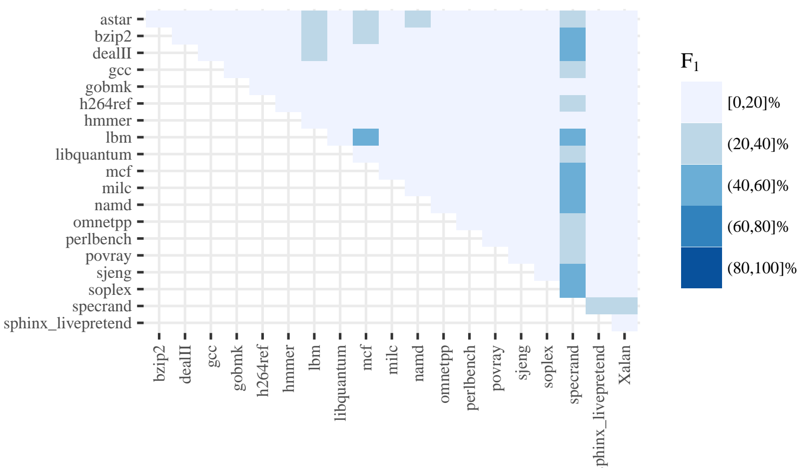


Figure 7.27: Quality (F_1 Scores) of Alignments of `CPU2006` Executable Pairs using Unscaled Relative Similarity

relative position in the executables. Figure 7.26 shows a graphical view of the stabilities. In the plot, no clear structure emerges.

Alignment Quality The alignment quality (F_1 scores of unscaled relative similarity alignments) ranges from 0.19% to 59%. Figure 7.27 shows a plot of these scores. In it, the alignments with `specrand` show a significant increase in alignment quality. These are due to the `specrand` executable containing very few functions, and the ability of the sequence alignment to apparently pair these well to other short executables. The best performing alignment is between `lbm` and `mcf`.

Figure 7.28 shows the alignment using raw function lengths between `lbm` and `mcf`. Both executables show the structures of the dynamic linking of the C library between alignment function indices 0 to 30 and 60 to 100 (see Occurence 1a). Since both executables contain only about 30 other functions, and the usage of the C-library functions follows common patterns, the complete alignment is dominated by the alignment of the wrapper functions for the dynamic linking, and although the executables contain different applications, the quality of the alignment is not bad.

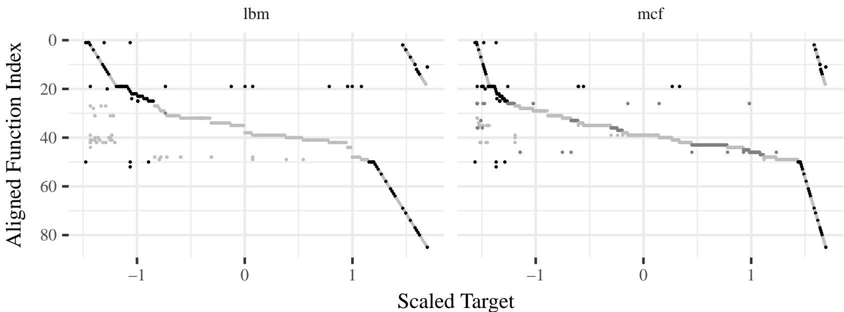


Figure 7.28: Alignment of `lbm` and `mcf` using Unscaled Functions Lengths

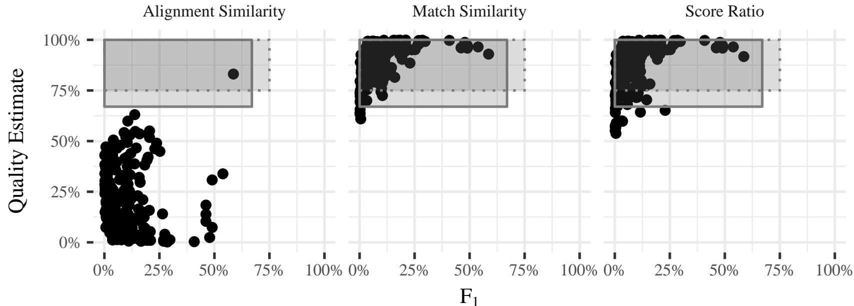


Figure 7.29: Correlation of Quality Estimates with F_1 Scores of `CPU2006` Executable Pairs

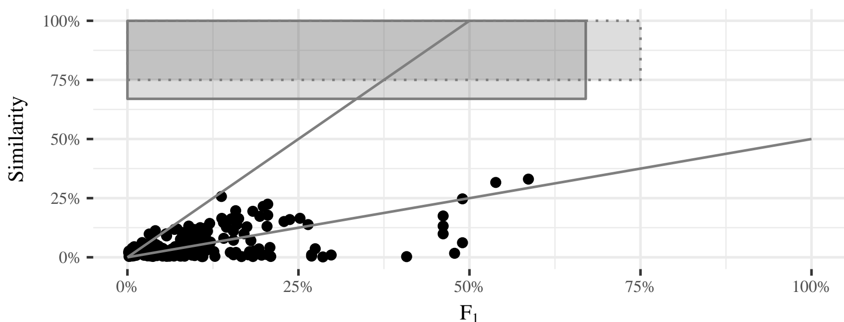


Figure 7.30: Correlation of Similarities with F_1 Scores of CPU2006 Executable Pairs

Quality Estimate Figure 7.29 shows the correlations between the *match similarity*, the *score ratio*, and the *alignment similarity* with the F_1 scores for the CPU2006 executable pairs. All alignments are of low to medium quality and only the *alignment similarity* reflects this as an estimate, and gives low to medium qualities for the alignments. The prediction scheme is valid, with only one outlier. Spearman's ρ is -0.15 with a significant p-value. This surprisingly negative correlation can be explained by recalling that no executables with high similarity are present.

Effectiveness Figure 7.30 shows a correlation plot between the F_1 score and the similarity. The plot does not show a significant correlation. Some alignments with low similarity have higher than expected F_1 scores (sector below the diagonal area). These are probably the very short alignments with the specrand executable. Spearman's ρ is 0.39 with a significant p-value. But, the values do not cover the whole range of the scales on which we want to asses the data.

The evaluation of this test case shows that the alignment was able to successfully align the few similar parts of smaller executables.

7.2.3 Validation

In this section, we investigate alignments calculated for executables from the test suite of Stojanovic, Radivojevic, and Cvetanovic [40]. Similar to our test suite, the validation test suite covers variations in the compiler and optimisation level, and dissimilar executables. It can serve as a validation, since these executables were externally built and cannot be chosen to perfectly fit out techniques.

STAMP							
Content	Stanford Transactional Applications for Multi-Processing						
Variation	Compiler, Optimisation						
	Min.	1Q.	Med.	Mean	3Q.	Max	
Similarity	2.4	7.2	14	23	33	94	
Stability	33	68	92	83	96	100	
Quality	0	0	1	19	31	99	
	ρ	p-value	>0.67	>0.75			
F_1 ˜ Aln. Similarity	0.69	$2 * 10^{-16}$	1224/1	1225/0			
F_1 ˜ Similarity	0.71	$2 * 10^{-16}$	1193/32	1207/18			

The compilers used in the creation of these executables are: CS: CodeSourcery ARM GNU/Linux tool chain, CW: CodeWarrior GCC for ARM, IAR: IAR Embedded Workbench, Keil: Keil ARM Compiler, SYS: SysProgs Prebuilt GNU Toolchain.

Similarity The similarities range from 2.4% to 94%, but most are low (third quartile is 23%). Figure 7.31 shows a plot of the similarities. The plot shows that higher similarities are only reached for executable pairs that stem from the same compiler, independently of the optimisation level used. The executable pairs which are compiled with SYS and CS compilers are of medium similarity. The executables compiled with IAR generally have low similarity, even when both executables are compiled with IAR.

Sequence Stability The stabilities (recall of function name identity alignment) range from 33% to 100% with a first quartile of 68%. These can be classified as high stabilities. Figure 7.32 shows a plot of the stabilities. All

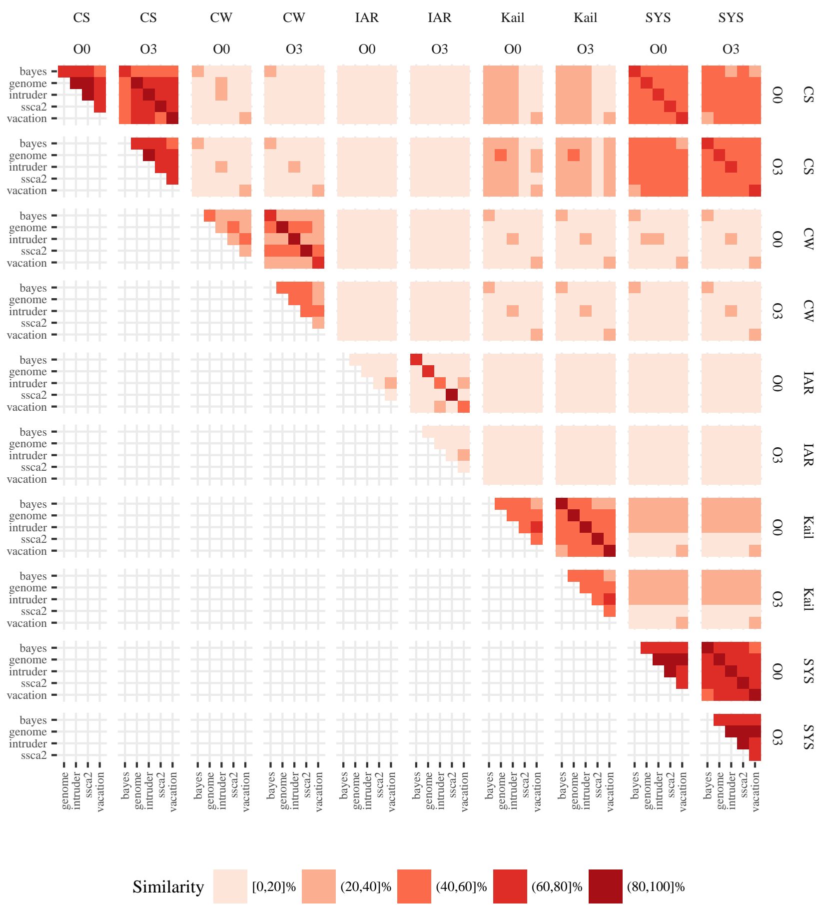


Figure 7.31: Similarity of STAMP Executable Pairs

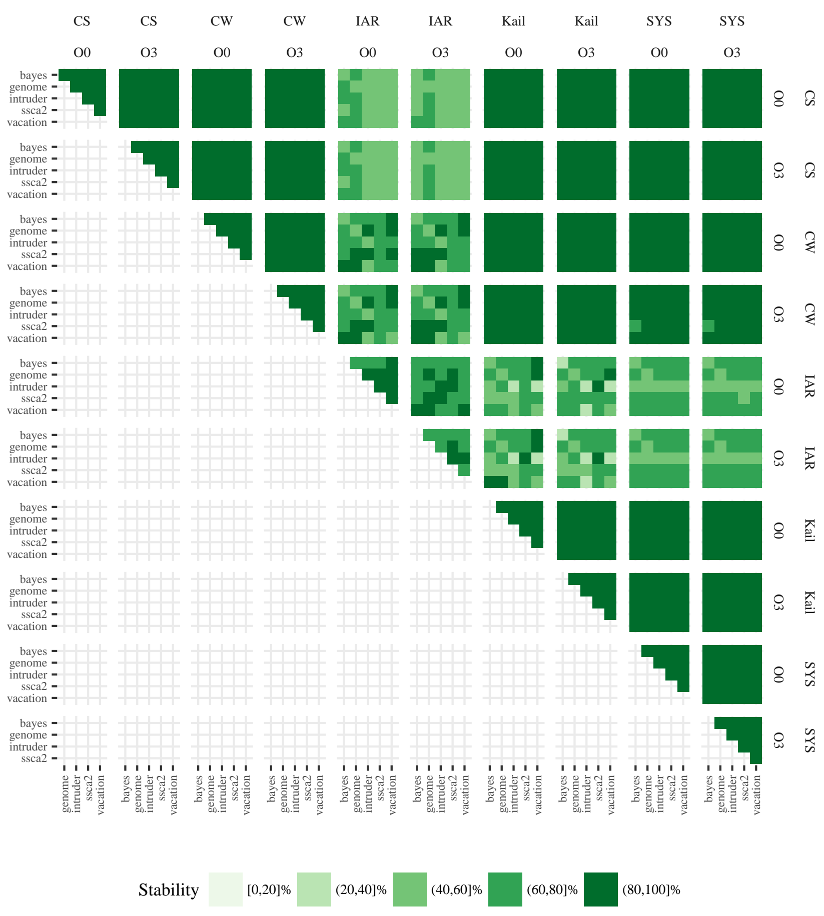


Figure 7.32: Stability (Recall of Name Identity Alignment) of STAMP Executable Pairs

stabilities in this plot are very high, except if the executables are compiled with the IAR compiler. These are of medium stability.

Alignment Quality The alignment quality (F_1 scores of unscaled relative similarity alignments) ranges from 0% to 99%, with a median of 1% and third quartile of 31%. These need to be classified as low quality. Nonetheless, in Figure 7.33, which shows the alignment qualities, the executables that share the same compiler and optimisation level (except of those compiled with IAR), can be identified as high-quality alignments. This roughly corresponds to the expectations from the similarity and stability, which already reduced the candidates for high-quality alignments to those executable pairs that share a compiler that is not IAR.

Quality Estimate Figure 7.34 shows the correlations between the *match similarity*, the *score ratio*, and the *alignment similarity* with the F_1 scores for the STAMP executable pairs. All estimates show clear correlations. The custom prediction scheme is valid with one outlier for the *alignment similarity*. Spearman's ρ is 0.69 with a highly significant p-value.

Effectiveness Figure 7.35 shows a correlation plot between the F_1 score and the similarity. The plot does not show a significant correlation. The plot shows alignments with a similarity that exceeds the achieved F_1 score (sector above the diagonal area). Additionally, there are several alignments that are counterexamples to the custom prediction scheme. Spearman's ρ is 0.71 with a highly significant p-value.

The evaluation of the first set of validation executables shows that the alignment can successfully align executables produced by most compilers, with the exception of executables from the IAR compiler (see Section 7.3). The alignment can also align the different elements (e.g., the `bayes` to the `ssca2` executables) of this test case.

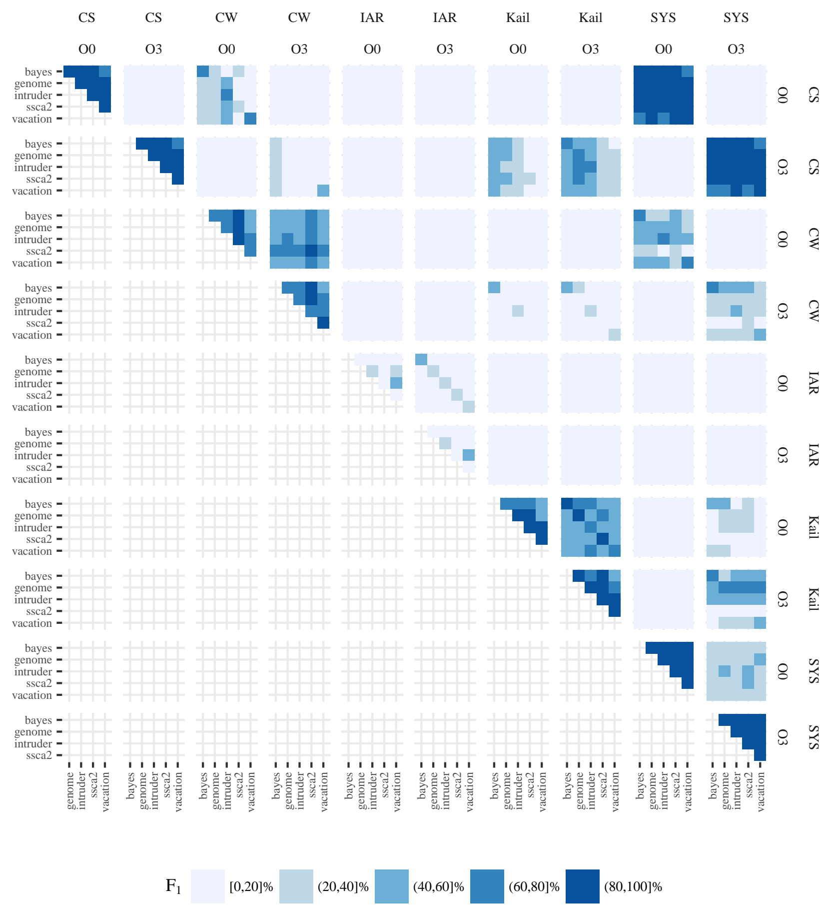


Figure 7.33: Quality (F_1 Scores) of Alignments of `STAMP` Executable Pairs using Unscaled Relative Similarity

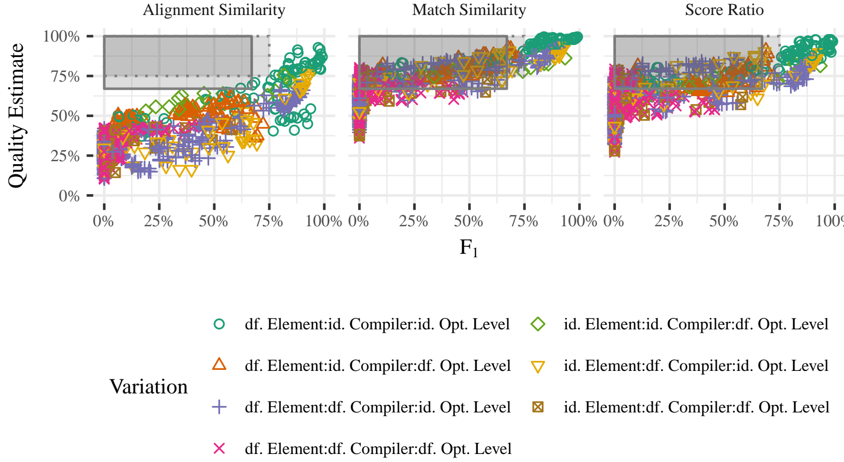


Figure 7.34: Correlation of Quality Estimates with F_1 Scores of STAMP Executable Pairs

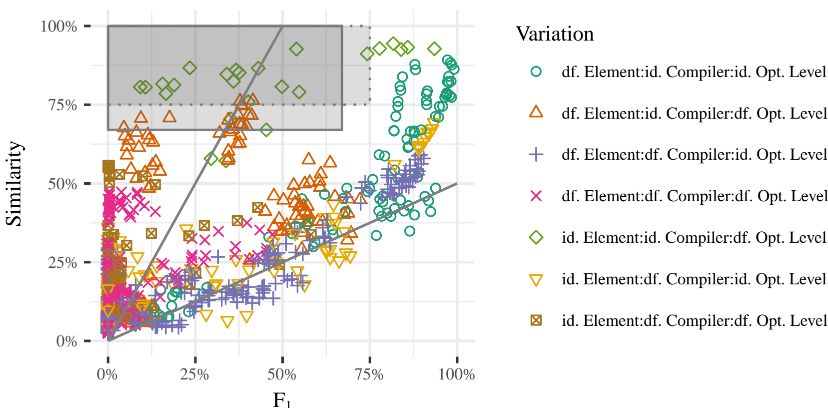


Figure 7.35: Correlation of Similarities with F_1 Scores of STAMP Executable Pairs

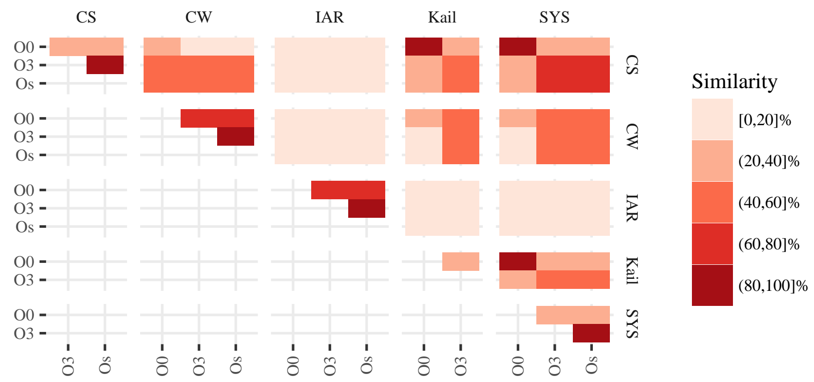


Figure 7.36: Similarity of `Busybox matrixssl` Executable Pairs

`Busybox matrixssl`							
Content	Common UNIX Utilities in a Single Executable						
Variation	Compiler, Optimisation						
	Min.	1Q.	Med.	Mean	3Q.	Max	
Similarity	9.8	14	25	34	47	93	
Stability	14	23	65	55	78	97	
Quality	0	0	6.9	20	38	85	
		ρ	p-value	>0.67	>0.75		
F_1 ˜ Aln. Similarity		0.64	$7 * 10^{-12}$	90/1	87/4		
F_1 ˜ Similarity		0.85	$2 * 10^{-16}$	84/7	84/7		

The compilers used in the creation of these executables are: CS: CodeSourcery ARM GNU/Linux tool chain, CW: CodeWarrior GCC for ARM, IAR: IAR Embedded Workbench, Keil: Keil ARM Compiler, SYS: SysProgs Prebuilt GNU Toolchain.

Similarity The similarities range from 9.8% to 93%, with a third quartile of 47%. These can be classified as low to medium. Figure 7.36 shows a

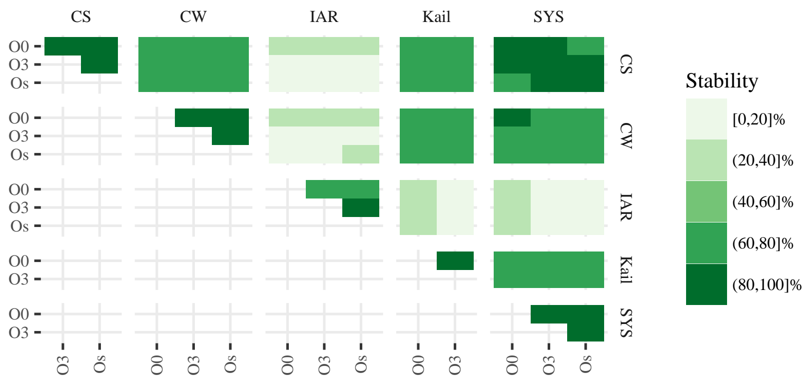


Figure 7.37: Stability (Recall of Name Identity Aligment) of `Busybox matrixssl` Executable Pairs

plot of the similarities. In the plot, most of the low similarities belong to executable pairs where one executable is compiled with IAR.

Sequence Stability The stabilities (recall of function name identity alignment) range from 14% to 97%, and are nearly evenly spread over the range (first quartile 23%, third quartile 78%). A clear classification is not possible. Figure 7.37 shows a plot of the stabilities. The lower stabilities can be identified as belonging to alignments where one executable is compiled with the IAR compiler.

Alignment Quality The alignment quality (F_1 scores of unscaled relative similarity alignments) ranges from 0% to 85%, with a median of 6.9% and third quartile of 38%. These need to be classified as low quality. Nonetheless, in Figure 7.38, which shows the alignment qualities, the executables that share the same compiler and optimisation level (except of those compiled with IAR), can be identified as medium quality alignments.

Quality Estimate Figure 7.39 shows the correlations between the *match similarity*, the *score ratio*, and the *alignment similarity* with the F_1 scores for the busybox matrixssl executable pairs. All estimates show positive correlations. The custom prediction scheme is valid with one and four

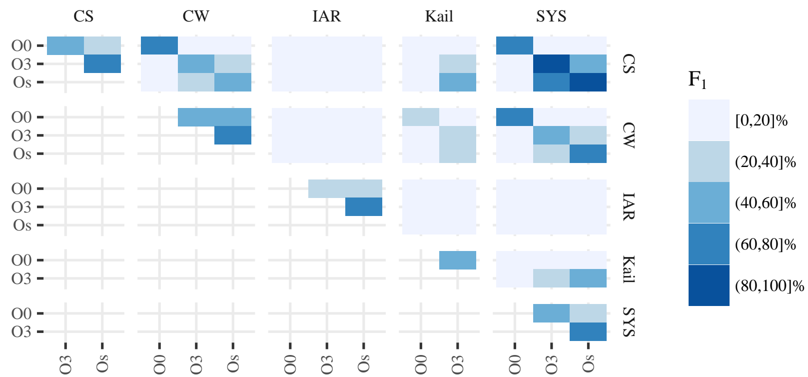


Figure 7.38: Quality (F_1 Scores) of Alignments of `Busybox matrixssl` Executable Pairs using Unscaled Relative Similarity

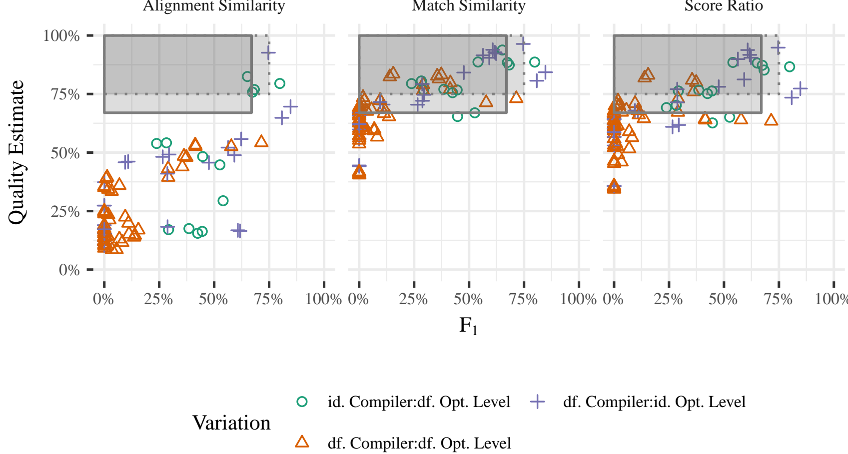


Figure 7.39: Correlation of Quality Estimates with F_1 Scores of `Busybox matrixssl` Executable Pairs

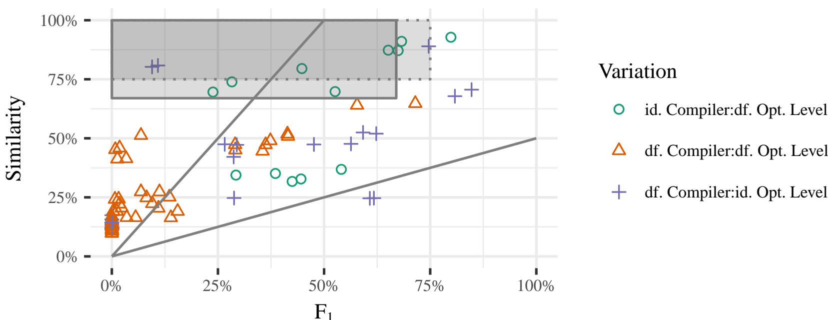


Figure 7.40: Correlation of Similarities with F_1 Scores of `Busybox` `matrixssl` Executable Pairs

outliers for the *alignment similarity*. Spearman's ρ is 0.64 with a highly significant p-value.

Effectiveness Figure 7.40 shows a correlation plot between the F_1 score and the similarity. The plot does not show a significant correlation. The plot shows alignments with a similarity that exceeds the achieved F_1 score (sector above the diagonal area), most of which are from pairs with different compiler and optimisation level. Additionally, there are several alignments that are counterexamples to the custom prediction scheme. Spearman's ρ is 0.85 with a highly significant p-value.

The second set of validation executables confirms the inability of the alignment to handle executables produced by the IAR compiler (see Section 7.3), but otherwise shows that the technique can produce alignments of acceptable quality.

With this set of executables, we have validated that our analysis can cope with different compilers and optimisations, and find similarities between otherwise dissimilar test suite elements. We have also shown that, if the analysis cannot produce high-quality results (alignments with the IAR compiler), the estimate can function as a warning indicator and mark such results.

Table 7.8: Summary of Evaluation Data in Terms of Variations, Similarity, and Quality

Name	Element	Version	Compiler	Optimisation	Platform	Architecture	Similarity	Stability	Quality	Effectiveness	Estimate
glibc	□	■	■	■	□	□					
RedBoot	□	▒	▒	▒	■	■					
musl	□	■	■	■	□	□					
vmlinux	□	■	□	□	□	■					
CPU2006	■	□	□	□	□	□					
STAMP	■	□	■	■	□	□					
Busybox matrixssl	□	□	■	■	□	□					

7.3 Summary

In this section, we summarise the evaluation data with regard to the questions posed at the start of the evaluation chapter.

In Table 7.8 we summarise the data from the evaluation. Each row represents one member of the test suite. The first set of columns shows the variation that this member covers. The second set of columns (similarity, stability) shows the ground truth information from the source-code relation (function names). The last set of columns (quality, estimate, effect) shows the data from the function size alignments using unscaled relative similarity. In the table, we use a ternary scheme for the presence of variations: ■ means the member varies in this attribute, □ means no variance, and ▒ means that no information is known about this attribute. The second and third notation are the same as used in the case studies.

The first question is, *to which degree the executable variations are order-preserving*. We can investigate this question by plotting the executable similarity and the sequence stability in a correlation plot.
If the variations would change the function sequence, even executables with

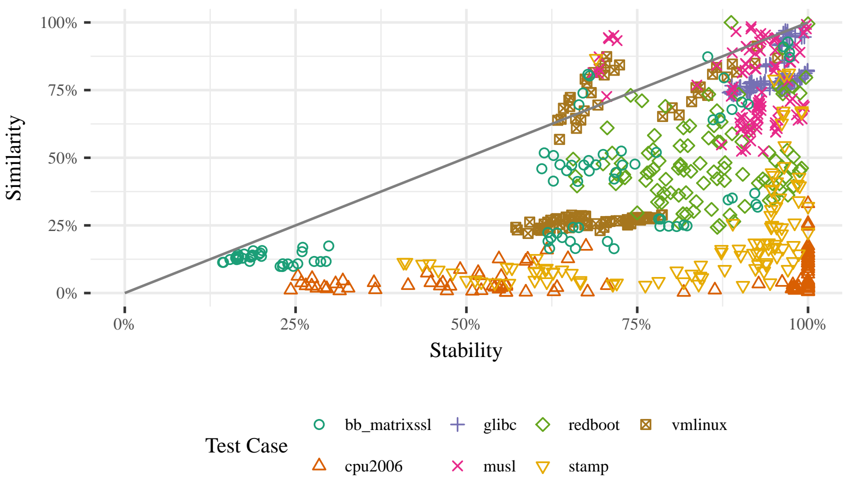


Figure 7.41: Similarity and Stability of a Random Sampling of 100 Function Name Identity Alignments per Test Case

high similarity could have permuted sequences, and therefore low sequence stability. If the variations do not change the sequence, one would expect an increase in stability for an increase in similarity.

The similarity and stability columns of Table 7.8 show that, except for the `Busybox matrixssl` case, the stability looks basically invariant to the similarity and is usually high. Figure 7.41 shows similarity and stability (recall) of a random sampling of up to 100 function name identity alignments per test case. In this plot, most points are in the lower-right triangle. This means that, as a general tendency, even for lower stabilities, the stability exceeds the similarity.

The second question is, *to which degree the differential analysis works for the order-preserving variations*. Since we know that the sequence stability generally exceeds the similarity, we can investigate this question by comparing the similarity with the achieved quality, and the number of counterexamples for our prediction scheme (effectiveness).

If the performance of the differential analysis depends on the order-preserving variations, one would expect a correlation between the similarity (and implicitly stability) and the achieved quality. If the performance of the differential analysis is independent of the order-preserving variations, executables with low similarity could result in high quality alignments and executables with high similarity could result in low quality alignments.

The similarity and quality columns in Table 7.8 show a general correlation of the two values. In the Effectiveness column of Table 7.8, we can see the number of counterexamples for the prediction scheme. This scheme highlights cases that have a high similarity, but where the alignment is not of high quality. Here, the `musl`, `vmlinux`, and `Busybox matrixssl` show visible numbers of counterexamples. For `vmlinux`, these stem from the underperformance of the differential analysis for inter-architectural alignments with the ARM32 architecture. For `musl`, these stem from the change introduced between versions 0.9.4 and 0.9.6. In both of these cases, the underperformance can be mitigated by selecting the scaled relative similarity function for the alignments. For `bb_matrixssl`, these cases are not easily identifiable.

Figure 7.42 shows a plot of the similarity versus the quality (F_1 score) of the unscaled relative similarity alignments. The plot, as well as Spearman's ρ on the data, which is 0.85 with a high significance, shows the general tendency of a correlation. In the prediction areas annotated in the plot, most of the counterexamples belong to `musl` or `vmlinux`, for which the underperformance can be mitigated by choosing another similarity function. The plot does not show a relevant amount of cases for which the quality exceeds the similarity.

Note that, although the alignment quality for executables compiled with IAR was generally low, these do not appear as malperformance, as the similarity was also low in these cases.

The last question is, *to which degree the differential analysis can estimate the quality of the returned alignment.* We investigate this question by comparing the proposed quality estimates with the achieved quality.

Figure 7.43 shows a random sampling of up to 100 alignments using unscaled relative similarity per test case. Both the *match similarity* and the *score ratio* show a left-upper triangular structure. In these cases, the estimate can roughly be interpreted as an upper bound on the achieved

Figure 7.42: Similarity and Quality (F_1 Scores) of a Random Sampling of 100 Unscaled Relative Similarity Alignments per Test Case

quality. For using the prediction as warning indicator for alignments of low quality, these measures are not useful. Contrary, the *alignment similarity* shows a correlation (Spearman's ρ is 0.85, with high significance). Since it resembles an increasing diagonal shape, it is usable both as an upper and lower bound on the achieved quality. The custom prediction scheme also does not show a significant number of outliers (compare with the estimate column in Table 7.8). This measure is therefore usable as warning indicator as well as positive predictor, where for *alignment similarities* larger than 67%, F_1 scores larger than 67% can be expected (analogously for 75% values). Also, the plot indicates an area without dots on the lower right, which indicates that for *alignment similarities* less than 25%, no F_1 scores larger than 50% exist.
Note that all evaluations of this these measures used the unscaled relative similarity function as parametrisation of the alignment.

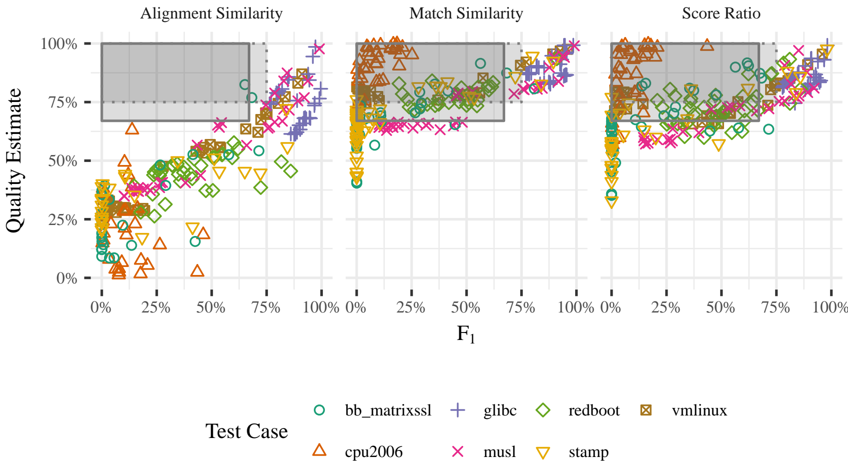


Figure 7.43: Quality Estimates and Achieved Quality (F_1 Scores) of a Random Sampling of 100 Unscaled Relative Similarity Alignments per Test Case

Threats to validity Since the differential analysis is backed by a quality estimate that warns about low quality alignments, the differential analysis main threat to validity are problems with this measure. The most prevalent drawback is a poorly chosen test suite, since the measure is only empirically validated. Such a test suite could involuntarily leave out cases in which the measure would be high for a low quality alignment. We mitigate this problem by also including executables into the test suite that, while covering part of our variation domain, are chosen and compiled by a third party.

The following threats apply to both to the structural as well as the differential analysis. A major limitation of the test suite is that it only covers software that is compiled from C and C++ code. While other executables might behave similarly, executables from other source languages are completely uncovered. Additionally, the two analyses do not take countermeasures against any kind of obfuscation. An obfuscating compiler could rearrange an application code to fit into the coupling pattern of another

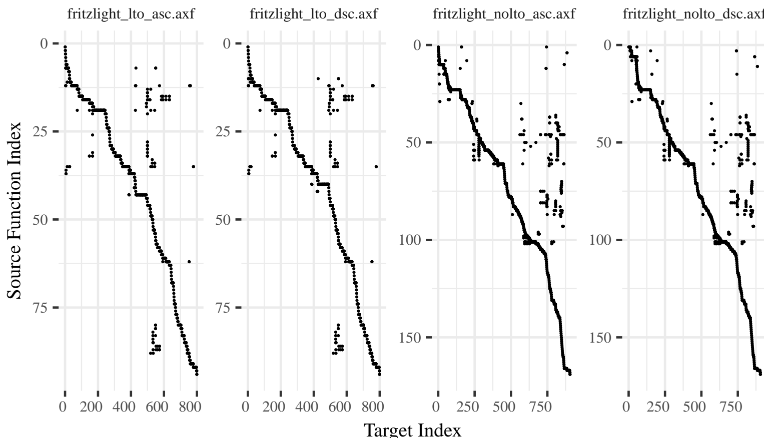


Figure 7.44: Link-Time and Non Link-Time Optimised Executables with Different Orders of Object Files in the Makefile

application (most easily by introducing dead code), or let the functions match a specific size profile by reordering or simply padding. Nonetheless, the author has performed these analyses on a malicious version of `putty` (in [45]) and a malicious version of `PokemonGO` (unpublished), and neither one takes any countermeasures, apart from hiding a few function names.

A major concern is *link-time optimisation*, which affects the interpretations of the plots and the stability of the sequences. Such an optimisation can indeed break the stability relation between the order of functions in the source files and the executable. On the other hand, link-time optimisation adds another stability to the executable's function sequence, because it encodes its optimisation and therefore sorting criteria into the executable. We investigate this behaviour with a short example.
Figure 7.44 shows the same version of an ARM embedded executable with and without link time optimisation and for ascending and descending order of object files mentioned in the top Makefile (no sorting in the library Makefiles) (the executable here is FritzLight[1]). The first observation is that

[1] *FritzLight*. URL: `https://github.com/arnew/fritzlight` (visited on 05/19/2017).

link-time optimisation, as expected, allows dead-code elimination to reach its full potential and to remove a large portion of the executable's functions. Additionally it changes most of the visible structures in the executable. The functions from files with different sort order are represented as functions 0 to 15 in the link-time optimised files and as functions 0 to 30 in the non link-time optimised files. For the non link-time optimised executables, the permutation of the files is visible in the representation, but for the link-time optimised case, the optimisation has actually further stabilised the order of functions, and thereby limited the effect of the permutation.
In summary, the presence of link-time optimisation does not prohibit the application of our patterns to the plot or the application of the differential analysis when both executables are link-time optimised, but it will change the function positions in the executable, and therefore does not allow drawing conclusions on the modular structure from locality information.

Another concern is the behaviour of the analysis on executables from the IAR (IAR Embedded Workbench) compiler. In both the `busybox matrixssl` as well as the `stamp` cases the alignments of similarities, stabilities, and alignment qualities for alignments between executables from other compilers and the IAR executables are low. Only the alignments of identical elements with different optimisation levels where both executables originate from IAR produce medium quality. In the following we inspect the alignments using function names and relative similarity on unscaled function sizes for three selected pairs of executables from the `busybox matrixssl` case. These pairs are two executables produced by the CW (Code Warrior) and the IAR compiler with optimisation levels O0, two executables produced by CW with optimisation levels O0 and O3, and two executables produced by IAR with optimisation levels O0 and O3.

First, we inspect the alignments on function names. Figure 7.45 shows three aligned executable pairs in three facets. Each facet displays the data of two executables in red and blue, in the same order as described above. For all facets, the executable written on the top is the blue executable. It is immediately visible that shapes of the plots of the executable produced by IAR are dissimilar to those produced by CS. Comparing the number of aligned pairs (derived from the vertical scale: aligned function index), it is clear that, between the different compilers, only few function names appeared in the same order, and the alignment had to create about 12000 pairs from the about two times 6000 input functions. The CS executable

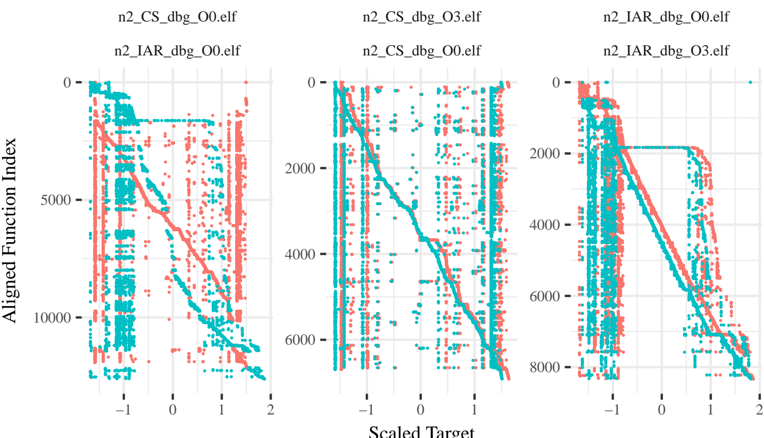


Figure 7.45: Alignments using Function Name Identity for Selected Executable Pairs from `Busybox matrixssl`

pair is shown mostly for comparison, shows a high quality alignment, and does not show noteworthy specialities. Last, the IAR executable pair results in about 8000 functions, significantly more than the number of functions in each of the pairs. This shows that between the different optimisation levels, the compiler changed the functions, and also the names of the functions.

Secondly, we inspect alignments of the same executable pairs as above but using the unscaled relative similarity function. Figure 7.46 shows the same facets and executables as described above. Here, the alignment between the different compilers is significantly shorter, but since the functions do not appear in the same order, obviously of low quality. The CS executable pair looks roughly similar between the ground truth and function size similarity alignments, showing a good alignment quality. Lastly, the IAR executable pair shows a shorter alignment than the ground truth alignment. This could mean that either the compiler changed function names and the functions are actually identical, but looking closely at the start of the sequence, the function name alignment shows a lack of blue functions, in the first few hundred executable pairs, and then continues with a very similarly

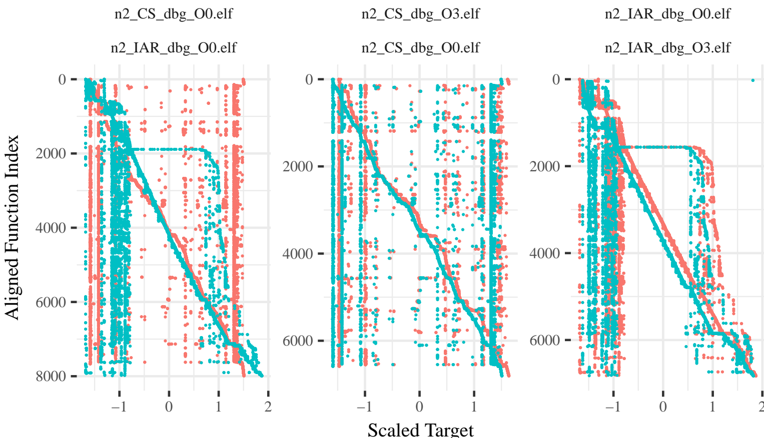


Figure 7.46: Alignments using Unscaled Relative Similarity for Selected Executable Pairs from `Busybox matrixssl`

shaped part in both plots. In the function size based alignment, the blue functions are basically smeared in this area. This shows that the function size similarity could not separate the functions sharply enough.

In summary, the IAR compiler uses a different code generation scheme compared with the other compilers in the evaluation. This prevents alignments between this compiler and other compilers to achieve a high quality. Nonetheless, the IAR executables are similar to each other, and the alignment can achieve medium quality in these cases.

8 Conclusion

In this chapter, we summarise the thesis and reflect on the contributions and goals from the introduction.

The goal of this thesis is providing a visually intuitive analysis that highlights certain control-flow structures, which may aid an analyst during first-time inspections for program comprehension of executables. Additionally, it sets out to identify similarities and differences in executables, enabling the transfer of analysis results between executables and parallel analyses of executables. The foundation of the following work is an order-sensitive representation called ordered control-flow graph (OCFG), along with two views for single executables and executable comparisons. While this representation was originally targeted to executables compiled from C and C++ code from embedded systems, it is generally applicable to any control-flow model that groups code into functions and uses a one-dimensional order to locate code.

Structural Analysis To address the first goal, we provide a new way of abstracting control dependences at statement level to control dependencies between pairs or sets of functions. This abstraction is realised by nine visual patterns that exploit different properties of the view of the OCFG.
With this analysis, we providing a visually intuitive way of identifying key locations of an executable that significantly reduces the amount of information an analyst needs to memorise, which makes a direct analysis challenging.

We evaluate the patterns on a diverse set of executables and in a detailed case study. While the application of the patterns is not targeted at finding each pattern in every executable but on a wide range of executable types, we locate patterns in each of the executables. Additionally, we validate each of the pattern applications using information from the source-code relation. In the detailed case study, we locate six of the nine patterns, which cover all of the functions present in the executable. The information from the patterns describes most of the key functionality of the executable.

The evaluation shows that the patterns can be found in the targeted executables, and the information to aid program comprehension can be found using the patterns.

Differential Analysis To address the second goal, we provide an algorithm for identifying similarities in pairs of OCFGs. Practically, this algorithm parametrises Hirschberg's algorithm for calculating sequence alignments using similarity measures for function pairs based on function sizes. Additionally, we propose estimate measures of the alignment quality that do not depend on external information.
With this analysis, we can identify similarities in executable pairs that contain several order-preserving variations. The quality estimate allows a reflection on the achieved quality and aids in selecting an appropriate parametrisation of the algorithm.

We evaluate the algorithm and the proposed parametrisations in terms of stability of the order-preserving variations, the achieved quality, and the proposed quality estimates by application of the algorithm on three sets of executables: One set of similar executables that contains order-preserving variations, one set of dissimilar executables, and one set of validation executables. The detailed evaluation is performed using a single parametrisation that ranked highest for most of the test cases. Where useful, the result of a better performing alternative is inspected as well. The order-preserving variations lead to executable pairs with high sequence stability, which shows that these variations indeed do not change the function sequence in the test cases. The achieved quality of the alignments is high, if the executable pairs are actually similar, but can degrade both for executable pairs of low similarity and for some variations that are not ideally covered by the proposed parametrisations. Lastly, one of the proposed quality estimates is able to both numerically predict the alignment quality and to function as a warning indicator (above or below 67%) that signals potentially low-quality alignments.

The evaluation shows that the analysis can indicate the achieved quality, and, in cases of good quality, that the similarities in the function sequences can be found. In summary, the differential analysis enables an analyst to transfer results or perform parallel analyses.

In addition to the theory, analyses, and evaluations described above, we provide an implementation of the OCFG, the graphical views, and the alignment.

In summary, our contributions successfully address several of the challenges of analysing executables at low representation level, insofar they allow

an intuitive orientation in the code for program comprehension and the exploitation of similarities between executables.

Future Work Based on the presented analyses and data, we propose several directions of future work.

Surprisingly, many of the variations that occur in executables are order-preserving. The evaluation highlights only three changes which broke this assumption, namely a change in `musl` that enables or disables the inlining of several hundred functions, the ARM32 variant of `vmlinux` which is not well alignable to the other architectures, and the validation executables compiled with IAR. A more thorough evaluation of real-world reverse engineering examples, like pairs or triples of executables sampled from actual embedded systems, could help to assess whether such cases actually occur. It would also be interesting to include obfuscated executables to inspect whether the obfuscations have an effect on sequence stability.

The estimate of the alignment quality is used as a warning indicator on the analysis result, but for the cases of our test suite, it also correlates with the executable similarity. With an evaluation that also covers random similar and dissimilar executables, it can be validated and then used to generally compare executables for similarity. A major application could for example be the fast inspection of malware similarities.

The structural analysis does not automatically detect and highlight the patterns. A simple variant of an automatic detection by the author is published [47], but the detection quality of it is very low. A more sophisticated image recognition approach could allow a wide-spread identification of the patterns, as well as an automated evaluation.

Currently, all of the proposed patterns are manually validated. With a test suite that contains ground truth about the design information (such as, for example, top-level modules, modularisation, libraries, and key-functions) and an automated detection of the patterns, the proposed patterns could be evaluated and validated further.

The differential analysis did not ideally perform in all cases. In cases with higher similarity and sequence stability, the function-size data was not sufficient for the alignment. Generally, other research in this area uses vastly more complex and sophisticated comparisons between the functions, and our analysis mostly relies on the position information in the sequence. But,

the alignment technique also allows the use of more complex comparison functions. In a preliminary work on sequence alignment [46], the author uses a vector of metrics for each function to map each function to an element of a finite alphabet and applies the standard protein sequence alignment techniques. Using similar data and more sophisticated comparison functions, it could be possible to achieve higher quality alignments, even for variations that have huge effects on function sizes.

Another limitation of this work is that the control-flow information the analyses are built upon are neither sound nor complete. While the evaluation shows it is possible to achieve high quality results with this data, it would be interesting to show how an evaluation based on other control-flow data, originating from more fine-grained analyses or perhaps traces, alters the results. For the structural analysis, we expect this data to improve the quality, since the analysis no longer contains dead code, but for the differential analysis, we expect the quality to degrade, since with fewer functions, the function size signature information is reduced, and the alignment may not reach its current quality.

Bibliography

[1] Mamoun Alazab, Sitalakshmi Venkataraman, and Paul Watters. "Towards Understanding Malware Behaviour by the Extraction of API Calls". In: *Second Cybercrime and Trustworthy Computing Workshop*. IEEE. 2010, pp. 52–59.

[2] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. "CodeSurfer/x86—A platform for analyzing x86 executables". In: *International Conference on Compiler Construction*. Springer. 2005, pp. 250–254.

[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. "Byteweight: Learning to Recognize Functions in Binary Code". In: *23rd USENIX Security Symposium*. USENIX. 2014, pp. 845–860.

[4] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. "The Concept Assignment Problem in Program Understanding". In: *Proceedings of the 15th International Conference on Software Engineering*. ICSE. Baltimore, Maryland, USA: IEEE Computer Society Press, 1993, pp. 482–498.

[5] Ruven Brooks. "Using a Behavioral Theory of Program Comprehension in Software Engineering". In: *Proceedings of the 3rd International Conference on Software Engineering*. ICSE. Atlanta, Georgia, USA: IEEE Press, 1978, pp. 196–201.

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. "BAP: A Binary Analysis Platform". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 463–469.

[7] In Kyeom Cho, TaeGuen Kim, Yu Jin Shim, Haeryong Park, Bomin Choi, and Eul Gyu Im. "Malware Similarity Analysis using API Sequence Alignments". In: *Journal of Internet Services and Information Security* 4.4 (2014), pp. 103–114.

[8] Cristina Cifuentes. "Reverse Compilation Techniques". PhD thesis. Queensland University of Technology, 1994.

[9] James R. Cordy. “TXL: A Language for Programming Language Tools and Applications”. In: *Electronic Notes in Theoretical Computer Science* 110 (2004), pp. 3–31.

[10] Thomas Dullien and Rolf Rolles. “Graph-Based Comparison of Executable Objects”. In: *Proceedings of the Symposium sur la Securite des Technologies de l’Information et des Communications*. 2005.

[11] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts, USA: Addison-Wesley, 1995.

[12] Bernhard Ganter, Gerd Stumme, and Rudolf Wille. *Formal Concept Analysis: Foundations and Applications*. Lecture Notes in Artificial Intelligence. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[13] Kyoung Soo Han, Boo Joong Kang, and Eul Gyu Im. “Malware Analysis using Visualized Image Matrices”. In: *The Scientific World Journal* 2014 (2014).

[14] Kyoung Soo Han, Jae Hyun Lim, Boo Joong Kang, and Eul Gyu Im. “Malware Analysis using Visualized Images and Entropy Graphs”. In: *International Journal of Information Security* 14.1 (2015), pp. 1–14.

[15] Daniel S. Hirschberg. “A Linear Space Algorithm for Computing Maximal Common Subsequences”. In: *Communications of the ACM* 18.6 (1975), pp. 341–343.

[16] James Wayne Hunt and M. Douglas MacIlroy. *An Algorithm for Differential File Comparison*. Tech. rep. Bell Laboratories, June 1976.

[17] Johannes Kinder and Helmut Veith. “Jakstab: A Static Analysis Platform For Binaries”. In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 423–427.

[18] Ralf Lämmel, Rufus Linke, Ekaterina Pek, and Andrei Varanovich. “A Framework Profile of .NET”. In: *Proceedings of the 2011 18th Working Conference on Reverse Engineering*. WCRE. IEEE, 2011, pp. 141–150.

[19] Timothy C. Lethbridge and Nicolas Anquetil. “Approaches to Clustering for Program Comprehension and Remodularization”. In: *Advances in software engineering*. Springer, 2002, pp. 137–157.

[20] Vladimir I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals”. In: *Soviet Physics-Doklady* 10.8 (1966), pp. 707–710.

[21] Carola Lilienthal. “Komplexität von Softwarearchitekturen: Stile und Strategien”. PhD thesis. University of Hamburg, 2008.

[22] Panagiotis K. Linos, Philippe Aubet, Laurent Dumas, Yann Helleboid, Patricia Lejeune, and Philippe Tulula. “Visualizing Program Dependencies: An Experimental Study”. In: *Software: Practice and Experience* 24.4 (1994), pp. 387–403.

[23] Sven Mattsen, Arne Wichmann, and Sibylle Schupp. “A Non-Convex Abstract Domain for the Value Analysis of Binaries”. In: *22nd International Conference on Software Analysis, Evolution and Reengineering.* SANER. 2015, pp. 271–280.

[24] Sven Mattsen, Arne Wichmann, and Sibylle Schupp. “BDDStab: BDD-based Value Analysis of Binaries”. In: *The Fifth Workshop on Tools for Automatic Program Analysis.* TAPAS. 2014.

[25] Anneliese von Mayrhauser and A. Marie Vans. *Program Understanding – A Survey.* Tech. rep. CS-94-120. Department of Computer Science, Colorado State University, 1994.

[26] Robert McGill, John W. Tukey, and Wayne A. Larsen. “Variations of Box Plots”. In: *The American Statistician* 32.1 (1978), pp. 12–16.

[27] George A. Miller. “The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information”. In: *Psychological Review* 63 (Mar. 1956), pp. 81–97.

[28] Lakshmanan Nataraj, Sreejith Karthikeyan, Grégoire Jacob, and B. S. Manjunath. “Malware Images: Visualization and Automatic Classification”. In: *Proceedings of the 8th International Symposium on Visualization for Cyber Security.* VizSec. Pittsburgh, Pennsylvania, USA: ACM, 2011, 4:1–4:7.

[29] Saul B. Needleman and Christian D. Wunsch. “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins”. In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453.

[30] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivire Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Èdouard Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[31] Norman Ramsey and Mary F. Fernandez. "The New Jersey Machine-code Toolkit". In: *Proceedings of the USENIX 1995 Technical Conference Proceedings*. TCON'95. New Orleans, Louisiana: USENIX Association, 1995, pp. 24–24.

[32] Cornelis Joost van Rijsbergen. *Information Retrieval*. 2nd. Newton, MA, USA: Butterworth-Heinemann, 1979.

[33] Nathan E. Rosenblum, Xiaojin Zhu, Barton P. Miller, and Karen Hunt. "Learning to Analyze Binary Computer Code." In: *23rd National Conference on Artificial intelligence*. Vol. 2. 2008, pp. 798–804.

[34] Chanchal Kumar Roy and James R. Cordy. *A Survey on Software Clone Detection Research*. Tech. rep. 2007-541. Queen's School of Computing, 2007.

[35] Teresa M. Shaft and Iris Vessey. "The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension". In: *Information Systems Research* 6.3 (1995), pp. 286–299.

[36] Madhu K Shankarapani, Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. "Malware Detection using Assembly and API Call Sequences". In: *Journal in Computer Virology* 7.2 (2011), pp. 107–119.

[37] Janet Siegmund. "Program Comprehension: Past, Present, and Future". In: *23rd International Conference on Software Analysis, Evolution, and Reengineering*. Vol. 5. SANER. Mar. 2016, pp. 13–20.

[38] Software Engineering Institute, Carnegie Mellon University. *Software Product Lines*. URL: `http://www.sei.cmu.edu/productlines/` (visited on 05/19/2017).

[39] Charles Spearman. "The Proof and Measurement of Association between Two Things". In: *The American Journal of Psychology* 15.1 (1904), pp. 72–101.

[40] Sasa Stojanovic, Zaharije Radivojevic, and Milos Cvetanovic. "Approach for Estimating Similarity between Procedures in Differently Compiled Binaries". In: *Information & Software Technology* 58 (2015), pp. 259–271.

[41] Michael James Van Emmerik. "Static Single Assignment for Decompilation". PhD thesis. The University of Queensland, 2007.

[42] Cheng Wang, Jianmin Pang, Rongcai Zhao, and Xiaoxian Liu. "Using API Sequence and Bayes Algorithm to Detect Suspicious Behavior". In: *International Conference on Communication Software and Networks*. ICCSN. IEEE. 2009, pp. 544–548.

[43] Martin P. Ward. "Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations". In: *Science of Computer Programming* 52.1 (2004), pp. 213–255.

[44] Michael Spence Waterman, Temple Ferris Smith, and William A. Beyer. "Some Biological Sequence Metrics". In: *Advances in Mathematics* 20.3 (1976), pp. 367–387.

[45] Arne Wichmann, Sandro Schulze, and Sibylle Schupp. "Analyzing Malware Putty using Function Alignment in the Binary". In: *Softwaretechnik-Trends* 36.2 (2016), pp. 15–16.

[46] Arne Wichmann and Sibylle Schupp. "Matching Machine-Code Functions in Executables Within One Product Line via Bioinformatic Sequence Alignment". In: *5th IEEE Workshop on Mining Unstructured Data*. 2015, pp. 12–16.

[47] Arne Wichmann and Sibylle Schupp. "Visual Analysis of Control Coupling for Executables". In: *Softwaretechnik-Trends* 35.2 (2015), pp. 7–8.

[48] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. "No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations." In: *22nd Annual Network and Distributed System Security Symposium*. NDSS. 2015.

Tools

[49] *BDDStab*. URL: https://www.tuhh.de/sts/research/projects/bddstab.html (visited on 05/19/2017).

[50] *Binary Analysis Platform*. URL: https://github.com/BinaryAnalysisPlatform/bap (visited on 05/19/2017).

[51] *BinID2*. URL: http://www.phenoelit.org/BinID/index.html (visited on 05/19/2017).

[52] *CodeSurfer/x86*. URL: https://www.grammatech.com/products/codesurfer (visited on 05/19/2017).

[53] *FermaT: The FermaT Program Transformation System*. URL: http://www.cse.dmu.ac.uk/~mward/fermat.html (visited on 05/19/2017).

[54] *GNU Binutils*. URL: https://www.gnu.org/software/binutils/ (visited on 05/19/2017).

[55] Ilfak Guilfanov. *IDA Fast Library Identification and Recognition Technology (FLIRT Technology): In-Depth*. 2012. URL: http://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml (visited on 05/19/2017).

[56] *Hopper: The macOS and Linux Disassembler*. URL: https://www.hopperapp.com (visited on 05/19/2017).

[57] *IDA Pro: Interactive Disassembler*. URL: http://www.hex-rays.com/products/ida/index.shtml (visited on 05/19/2017).

[58] *Küstennebel*. URL: https://github.com/arnew/kuestennebel (visited on 05/19/2017).

[59] *Radare Reverse Engineering Framework*. URL: https://radare.org/r/ (visited on 05/19/2017).

[60] *retdec: Retargetable Decompiler*. URL: https://retdec.com (visited on 05/19/2017).

[61] *The Jakstab Static Analysis Platform for Binaries*. URL: http://www.jakstab.org (visited on 05/19/2017).

[62] *The TXL Programming Language*. URL: https://www.txl.ca/ (visited on 05/19/2017).

[63] zynamics. *BinDiff*. URL: https://www.zynamics.com/bindiff.html (visited on 05/19/2017).

[64] zynamics. *BinNavi*. URL: https://www.zynamics.com/binnavi.html (visited on 05/19/2017).

Testsuite

[65] *Doom*. URL: https://en.wikipedia.org/wiki/Doom_(1993_video_game) (visited on 05/19/2017).

[66] *FritzLight*. URL: https://github.com/arnew/fritzlight (visited on 05/19/2017).

[67] *Linux Kernel Source Tree*. URL: https://github.com/torvalds/linux (visited on 05/19/2017).

[68] *musl libc*. URL: http://www.musl-libc.org/ (visited on 05/19/2017).

[69] *RedBoot*. URL: https://sourceware.org/redboot/ (visited on 05/19/2017).

[70] *SPEC CPU^{TM} 2006*. URL: https://www.spec.org/cpu2006/ (visited on 05/19/2017).

[71] *The GNU C Library (glibc)*. URL: http://www.gnu.org/software/libc/ (visited on 05/19/2017).

[72] *Yaboot*. URL: https://web.archive.org/web/20160312033912/http://yaboot.ozlabs.org/ (visited on 05/19/2017).