Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Leibniz Universität Hannover

Dominik Töllner

Multivariant ELF Executables for Dynamic Variability via Address-Space Views

Masterarbeit im Fach Informatik

13. Oktober 2021

Please cite as:
Dominik Töllner, "Multivariant ELF Executables for Dynamic Variability via Address-Space Views" Master's Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, October 2021.

Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

Multivariant ELF Executables for Dynamic Variability via Address-Space Views

Masterarbeit im Fach Informatik

vorgelegt von

Dominik Töllner

geb. am 1. Februar 1995
in Langenhagen

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**Florian Rommel, M.Sc.**
Beginn der Arbeit:	**21. April 2021**
Abgabe der Arbeit:	**21. Oktober 2021**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Dominik Töllner)
Hannover, 13. Oktober 2021

ABSTRACT

Variability is the key concept to facilitate multivariance in computer applications. *Static variability* is applied at compile-time and allows developers to decide which parts of their software should be part of the final executable. Its benefit is a tailored binary file which only includes the data necessary to perform its tasks while omitting other data parts. A drawback is that not every decision can be made at compile-time which is why *dynamic variability* allows to defer that decision to the runtime of the application. During execution, it alters its behavior based upon its internal state to provide the desired variability. While this is a benefit, it also increases the runtime latency as the program has to evaluate its internal state each and every time. Even if the internal state has not changed.

This thesis bridges the gap between static and dynamic variability and tries to combine the benefits of both approaches while reducing their individual drawbacks. During this thesis a linker has been extended to allow building and embedding tailored application views inside the executable. With the help of a runtime library the threads can individually switch to these specialized views to benefit from static variability. The library creates dedicated *Address-Space Views* which contain the views to which the application threads can dynamically switch to during execution, thus combining static with dynamic variability. The evaluation shows that the established method can be applied to real-world applications. Additionally, it can provide a significant positive performance impact in certain scenarios. Individual tailored views also allow to harden security in multithreaded computer applications by locally restricting the code a thread can execute.

KURZFASSUNG

Variabilität ist das Schlüsselkonzept, um Multivarianz in Computeranwendungen zu ermöglichen. *Statische Variabilität* wird zur Übersetzungszeit angewandt und ermöglicht es Entwicklern zu entscheiden, welche Teile ihrer Software Teil der finalen Anwendung werden sollen. Ihr Vorteil ist ein zugeschnittenes Programm, welches nur die Daten enthält, die es zur Ausführung benötigt, während es andere Daten weglässt. Ein Nachteil ist, dass nicht jede Entscheidung zur Übersetzungszeit getroffen werden kann, weshalb *dynamische Variabilität* es erlaubt, diese Entscheidung auf die Laufzeit der Anwendung zu verlagern. Während ihrer Ausführung passt sie ihr Verhalten gemäß ihres internen Zustandes an, um die gewünschte Variabilität zu gewährleisten. Obwohl das ein Vorteil ist, erhöht es ebenfalls die Laufzeitlatenz, weil das Programm seinen internen Zustand stets erneut auswerten muss. Auch, wenn sich dieser nicht geändert hat.

Diese Abschlussarbeit schließt die Lücke zwischen statischer und dynamischer Variabilität und versucht, die Vorteile beider Ansätze zu vereinen, während sie ihre individuellen Nachteile minimiert. Der in dieser Arbeit erweiterter Binder ermöglicht es, zugeschnittene Sichten der Anwendung zu erzeugen, die zusätzlich in der Anwendung abgelegt werden. Mittels einer Laufzeitumgebung können Fäden während der Ausführung des Programms individuell zu diesen spezialisierten Sichten wechseln und so von den Vorteilen statischer Variabilität profitieren. Dabei erzeugt die Laufzeitumgebung dedizierte *Adressraumsichten*, welche die zugeschnittenen Sichten enthalten, zu welchen während der Ausführung gewechselt werden kann und verbindet so statischer mit dynamischer Variabilität. Die Evaluation zeigt, dass die erarbeitete Methodik auf Anwendungen aus der realen Welt angewandt werden kann. Zusätzlich dazu kann sie dabei einen signifikanten, positiven Einfluss auf die Performanz der Anwendung haben. Weiterhin erlauben es individuell zugeschnittene Sichten außerdem, die Sicherheit in mehrfädigen Computeranwendungen zu verstärken, indem der ausführbare Code eines Fadens und somit sein Handlungsspielraum lokal eingeschränkt wird.

CONTENTS

Abstract **v**

Kurzfassung **vii**

1 Introduction **1**

2 Fundamentals **3**
2.1 The Executable and Linkable Format . . . 3
2.1.1 Sections . . . 4
2.1.2 Segments . . . 4
2.2 Building an application: Compiler and Linker . . . 6
2.2.1 Symbols . . . 7
2.2.2 Relocations . . . 8
2.2.3 Memory overlay . . . 10
2.3 Address-Space Views . . . 11
2.4 Related work . . . 12
2.5 Summary . . . 14

3 Architecture **15**
3.1 Expanding the LLVM linker . . . 15
3.1.1 Multivariant object files . . . 15
3.1.2 Deploying memory overlay . . . 19
3.1.3 Local Symbols and Relocations . . . 21
3.1.4 Preparing data for the runtime . . . 22
3.2 Incorporating a runtime . . . 22
3.3 Summary . . . 24

4 Evaluation **27**
4.1 Synthetic test cases . . . 27
4.2 Memcached as a real-world application . . . 29
4.3 Summary . . . 32

5 Discussion **35**
5.1 Increased code complexity in the linker . . . 35
5.2 Programming interface . . . 35
5.3 Executable size overhead . . . 36
5.3.1 Meta data and view data . . . 36

5.3.2 Redundancy and view-local data 37
5.4 Performance impact 37
5.5 Future work 39
5.6 Summary 40

6 Conclusion **41**

Lists **43**
List of Acronyms 43
List of Figures 45
List of Tables 47
List of Listings 49
Bibliography 51

Appendix **53**

1 INTRODUCTION

Modern computer applications provide a huge amount of configurability. *Static variability* allows developers to configure their applications while the source code is translated into binary code. The results of different configurations are different executables of the same code base which are tailored to the exact needs of their use-cases. The benefits depend on the configurations taken, but can be faster execution, smaller binary and memory size or increased portability. A big problem of static variability is, that it does not cover the whole space of decisions a developer or application can make. Some decisions can only be made while the application is running, for example to check if a mouse is installed on the target system. For such use-cases *dynamic variability* defers that evaluation to the runtime of the application to allow it to configure itself properly for every target platform. Dynamic variability can be expensive if certain decisions are evaluated all over again even though the system's internal state did not change. Taken the example above an application typically needs to check if a mouse is installed only once, however it is in charge of re-evaluating that condition every time it wants to access the mouse to ensure it is available. A more desirable approach would be to roll out two versions of that application - one providing mouse support and one which is keyboard-only, because that reduces image size and runtime latency overhead by omitting conditional checks and application code. Since the set of different application versions rises exponentially with the occurrence of new configuration options the developers will stick to the non-tailored version which configures itself at runtime.

This thesis tries to combine both approaches to allow developers to create tailored versions of the application, but let the application decide at runtime which version it will use. The idea is to embed these different versions, further called *views*, in the executable in order to switch between these views during its runtime. Developers should be able to describe certain program code as multivariant code to generate views during the building of the application. A linker will be extended to process multivariant object files which contain the different versions the programmers of the target program. It will make use of *memory overlaying* where multiple application sections are linked and relocated towards the same virtual addresses. The result of memory overlaying are sections which reside at the same memory location and can be swapped in and out by a memory overlay manager. That overlay manager will be developed during the course of this thesis too, and it makes use of *Address-Space Views (ASVs)* to create distinguished application views. Address-Space Views allow threads of the same process to diverge in their execution code and provide the technical interface of applying runtime multivariance in this approach. Since these views give developers the option to restrict the code of certain threads they can additionally be leveraged to increase security as Section 5.3.2 and Section 5.5 will explain.

Both, the extended linker and the developed runtime overlay manager, are evaluated in synthetic test cases as well as in the real-world application *memcached*.

This thesis is structured as followed. Chapter 2 explains the concept of the Executable and Linkable Format (ELF) and introduces the reader into sections and segments. In order to properly build the sections the linker makes use of symbols and relocations which are discussed afterwards. Memory overlaying represents the final part of the application building process and examines how multiple program parts can be structurally lied on top of each other. After that the fundamentals shed light on the functionality of ASVs to let the reader understand the technical concept behind the developed runtime application multivariance. Related work closes the chapter by comparing existing research to this thesis in order to figure out similarities and differences. In Chapter 3 the architecture of this work is described as the linker's source code base has been extended to link and deploy multivariant executables. The building of a runtime overlay manager completes the chapter and reveals how ASVs are configured to include the different tailored versions of the target program. To measure the effects of the taken approach Chapter 4 evaluates the developed architecture in several synthetic test cases as well as in memcached. Chapter 5 discusses the measurements and explains the limitations of this thesis and gives several suggestions on how to continue the project in future work. In the last chapter the reader will be presented a conclusion highlighting the main takeaways and lessons learned to summarize how the established approach bridged the gap between static and dynamic variability.

FUNDAMENTALS 2

To fully understand how the concept of multivariance in computer applications has been established within this thesis, the reader will be introduced into several basic topics which describe what ELF files are and how they are built, how memory overlaying at the linker- and runtime-level works and how specific Address-Space Views are utilized to facilitate this runtime memory overlaying. After that related work is presented which describes similarities and differences between this thesis and other existing research. The summary recapitulates lessons learned within the fundamentals to depict the most important parts of all concepts.

2.1 The Executable and Linkable Format

When a computer application is loaded into the main memory to start its execution the entity which loads the executable, namely the *loader*, has to be provided with some information about the executable in order to prepare its execution. The loader has to know which parts of the binary must be loaded, needs to allocate enough memory, has to resolve dynamic runtime dependencies to other shared objects and must transfer control to the application properly. In order to pass this information to the loader a common language, a *format*, is necessary which the loader is able to parse and understand.

The Executable and Linkable Format describes how to organize the aforementioned information in an application. The *compiler* and *linker* both use it to store the application's data in an image file in order to pass it to the loader. As the name suggests ELF files do not only represent executable binary files. It does also describe object files which are parts of the main application that cannot run on their own and have to be linked together or dynamic libraries such as shared objects[18, p. 7]. The ELF is a very common executable file type under Unix operating systems, but it is not limited to a specific system or processor architecture[6]. In order to determine the type of a file a common technique is to place so-called *magic bytes*, often at the beginning of a file[19]. These magic bytes should uniquely identify the format of the current file and therefore hinting how to parse it. Keep in mind that these *magic bytes* are only **recommendations** - a file can have the magic bytes of an ELF but must not be formatted as such, although it is common sense that file formats match their respective magic bytes.

While the loader parses the executable it will find further metadata within the file itself describing its contents. They are composed as the **ELF header** and the most important properties are the required architecture, a starting address and file offsets for different tables such as the Program Header Table (PHT) or the Section Header Table (SHT)[12]. These two tables are of special interest as they have been expanded within this thesis to support a new approach of multivariant applications.

Each of these tables holds information about different entities of the program. The SHT holds information about *sections* while the PHT contains information about *segments*.

2.1.1 Sections

A *section* is an aggregation of related data in a compound manner. Simply spoken sections incorporate data which belongs together. When a program is developed it consists of multiple types of information - there is actual code to execute, data that this code refers to and other resources. Therefore, the program includes sections where the actual code is placed and other sections where the actual data is placed. It is a separation of concern and every section has its own meaning within the application.

All these sections come with different attributes describing its contents. This is necessary to further distinguish the aforementioned types of information. When a developer writes code it is common upon programming languages to allow constant and non-constant data. Given this practical example, this type of information has to be transported and retained during the runtime of the application. Therefore, the ELF must be able to represent constant and non-constant data. Section attributes allow to model this behavior and thus additionally classify its content, as they transport this information to the loader which allocates memory with respect to these attributes. Other attributes describing sections are its address during the runtime, the file offset within the ELF or its size. The ELF does not enforce specific names for sections, however a common set of names have been established which include:

- **.text:** Aggregates data about program code to execute
- **.init:** Aggregates data about program code to execute for initialization
- **.data:** Aggregates globally defined data
- **.rodata:** Aggregates globally defined constant data

Even though sections transport information to the *loader* it's not the section itself which is of particular interest to the latter. For the successful execution of the application section information on their own is not relevant. In fact the information about sections is not used by the loader at all later on. So what is the reason for sections after all? They still aggregate related data. But the sections themselves are then aggregated to *segments*. The chain of thought is illustrated in Figure 2.1. Sections aggregate data of equal semantic meaning and segments aggregate sections. The entity which aggregates sections in segments is called the *linker*. How the linker does that and how it decides which sections belong to which segments will be explained in Section 2.2.

2.1.2 Segments

Segments are the part of organizational structure which tells the *loader* how and what to load into the main memory in order to prepare the execution of the application. This is the information that the loader uses and why the section information on its own is irrelevant to the loader. Segments have attributes being similar to those of the corresponding sections. These include, but are not limited to, access rights such as *read, write* and *execute*, a size, a file offset, a virtual address and others. In other words segments only really do describe a continuous space of memory and sections are part of that memory space. The loader is able to get the information about all those segments and therefore makes several decisions. It allocates memory space big enough to fit the size of the segment, tries to load the specific segment at the specific Virtual Address (VA) and sets the virtual memory page access bits according to the access rights specified in the segment.

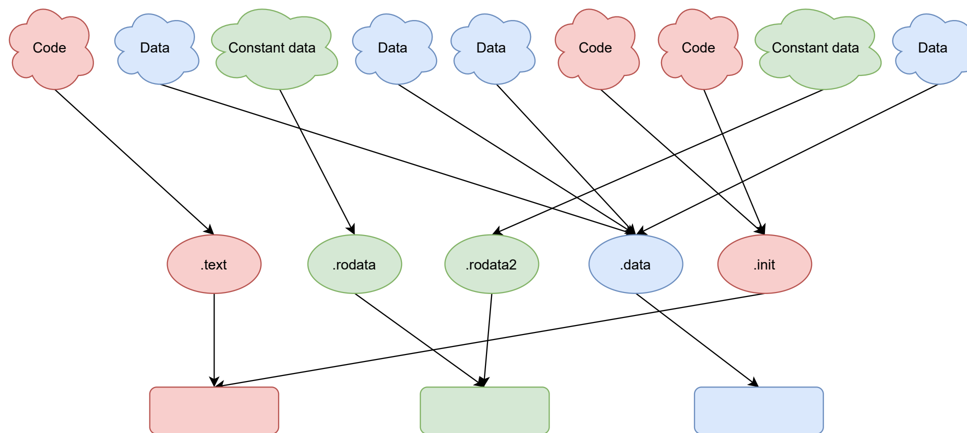


Figure 2.1 – Chain of thought for related data, sections and segments. Data that belongs together is aggregated into sections. Sections are then aggregated into segments. Segments do typically not have a name associated. Clouds represent types of data, circles represent sections and squares represent segments.

That means that all sections inside a segment share the same access rights which is why the linker places sections of equal access rights into the same segment and creates new segments for those requiring other types of access. This explains why the loader does not care about sections directly. Its only task is to acquire memory, place the segments appropriately in memory, eventually resolving some dynamic dependencies and transfer control to the program. Figure 2.2 summarizes and displays the relationship between *sections* and *segments* and gives an example of how they are laid out within a typical ELF file.

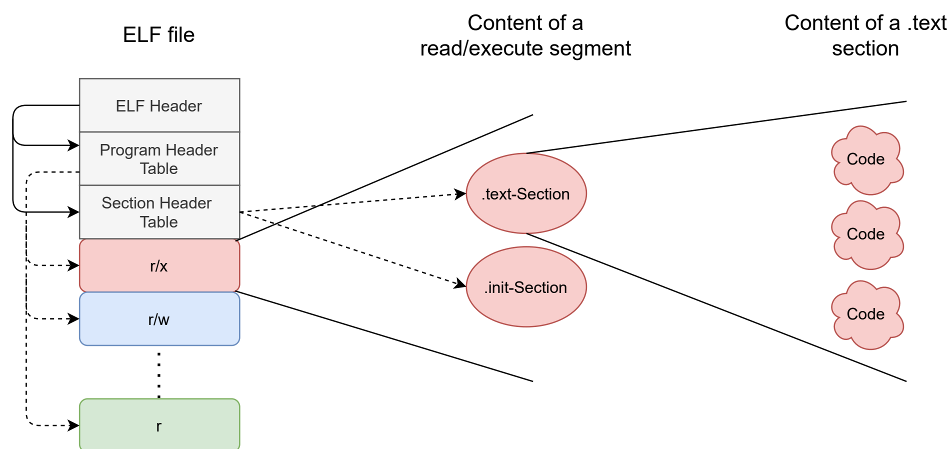


Figure 2.2 – An overview of an ELF and its content. It starts with a header further describing its different types of data such as the PHT and the SHT. The PHT holds information about all *segments* in the ELF. Segments do not have a name associated, but can be distinguished by their access rights: $r = read, w = write, x = execute$. Within a segment different *sections* are placed. These sections aggregate information of equal semantic meaning such as code or data for example.

2.2 Building an application: Compiler and Linker

At this point the reader got a basic introduction into what an ELF file is and how it is typically structured. Since one key aspect of this thesis is the expansion of ELF files to introduce a new mechanism of multivariant applications, it is crucial to understand how these files are actually build.

To build a computer application one possibility is to write code in a programming language and let a *compiler* translate that code into another language which the computer, or more precisely the execution unit of the computing system, is able to understand. The whole space of compilers is out of the scope of this thesis, but it will depict one part of compilers which describes how applications written in languages such as C or C++ can be translated into executable files.

Typically, the application's source code base consists of one or more files containing the code the application should execute at runtime. To build the application the compiler processes each file individually and translates it into an *object file*. This object file can be seen as a module of the application: It forms one part of the final program, but does not describe it completely on its own which is why it cannot be run. But all object files together fully describe the application so in conjunction they form the executable. The process of combining, or *linking*, object files is done by the *linker* and we will discuss it in a moment. The structure of such an object file depends on the system architecture to which it is compiled. For Unix systems they are organized as ELF files as explained in Section 2.1 and therefore object files contain sections as well. However, they do not contain segments for the whole reason that they are not executable, because the loader will not load them directly.

In essence the *compiler* takes all source code files of the application and translates them into *object files*. These object files are not executable on their own and have to be linked together by the *linker* to form the final executable binary as Figure 2.3 illustrates.

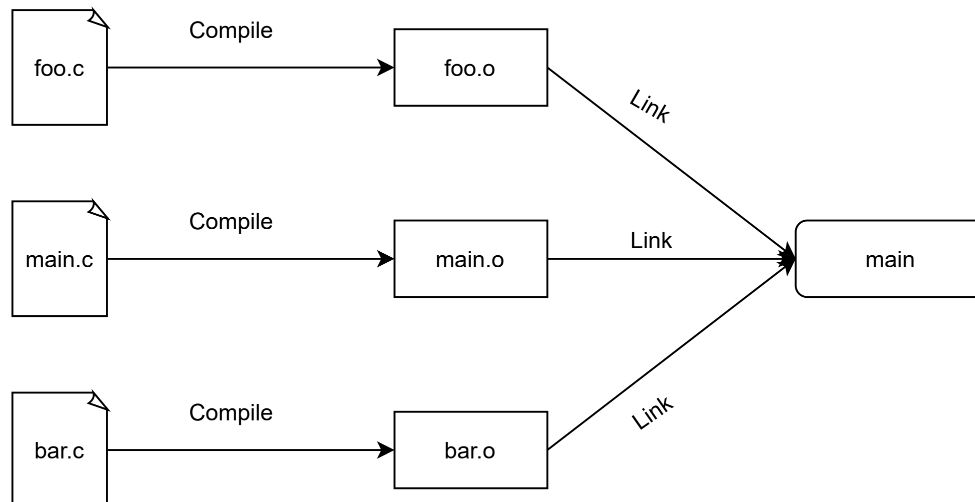


Figure 2.3 – Basic visualization of generating an executable file out of multiple input source code files. Every file is translated into an *object file* by a *compiler* and then linked together by a *linker* to produce the output executable.

So far so good, but the mindful reader will question how the compiler deals with references across multiple source code files. What does the compiler do when a function is defined in *foo.c.* but *main.c* wants to call it? We just examined that the compiler translates source code files file-by-file into object files. In fact, the compiler cannot be aware of the definition of a function within in another source code. In C/C++ this problem is addressed by the concept of *declarations* and *definitions*. A *declaration* of a data object (be it a function or just plain data) is the description of the **existence** of this data object. Simply spoken a declaration tells the compiler that there is this data object, but it does not actually say how it looks like e.g. does not define its value. The programmer just mentions the pure existence. A *definition*, however, is a *declaration* plus the information of how the data object really looks like. To circumvent the problem examined before the developer can *define* a function inside of *foo.c* and *declare* it in *main.c*. That way the compiler knows about the function in *main.c*, however it does still not know where it is defined, so a real call to that function will not succeed.

This is the introduction into the second entity of application building: the *linker*. The linker gets all object files as an input and creates the final output binary. Therefore, it works with **all** the data of the application whereas the compiler only operates on **one part** of the application, namely the source code of the file it currently processes. In the example above that means that the linker has access to the *main.o* file with the *declaration* of a function and the *foo.o* file with the *definition* of that same function. Because of that it is able to *link* the references in main.o of this function to its actual definition in foo.o. In essence it is the task of the linker to resolve references across multiple parts of the application to build an executable file. In the first step these references are unresolved within the individual object files. The compiler then places *symbols* and *relocations* within these files, so that the linker can find all those references and resolve them appropriately.

2.2.1 Symbols

Symbols allow the compiler to transport the information of unresolved references across the application's modules. They are organized in a symbol table and contain both, symbol definitions and symbol declarations[8]. The term *symbolic definition* describes the aforementioned function definition in one object file and *symbolic reference* describes the declaration of that function in another one. The symbol table gathers information about all symbols in a relocatable file, it aggregates data of equal semantic meaning. This is a concept which has been introduced already, and the outcome has been a *section* in which that data is placed into. Symbols are no exception to that and the common name of their section is **.symtab**. When the *linker* wants to find the symbol tables of the object files it goes a straightforward path. With the help of the ELF header and a specific magic value the symbol table can be found and that is shown in Figure 2.4.

During linking the linker iterates over every entry within the symbol table of an object file and connects *references* to their *definitions*. Since both of these types are hold in the same single symbol table, the linker has to identify them properly and this is done by the *symbol section header index* which denotes the section in which that symbol is defined. If a symbol has an index of the magic value *SHN_UNDEF* the linker knows that this symbol is not defined within its object file, hence it is a reference and not a definition, so it must resolve this dependency. It then searches for an object file in which this symbol is actually defined e.g. in which the *section header index* is a real index into an object files section. In order to find such a symbol these symbols have to be linkable, or *bindable*, to other object files. This property is defined by the *binding* of a symbol and while there are many different bindings the most basic decision is made between *local* and *global* symbols. Local symbols are symbols that are only bindable within the module which *defines* them. That means that if one module of the application defines a local symbol and another module references that symbol then the linker will abort since the symbol definition is only visible within the first module itself. Tables

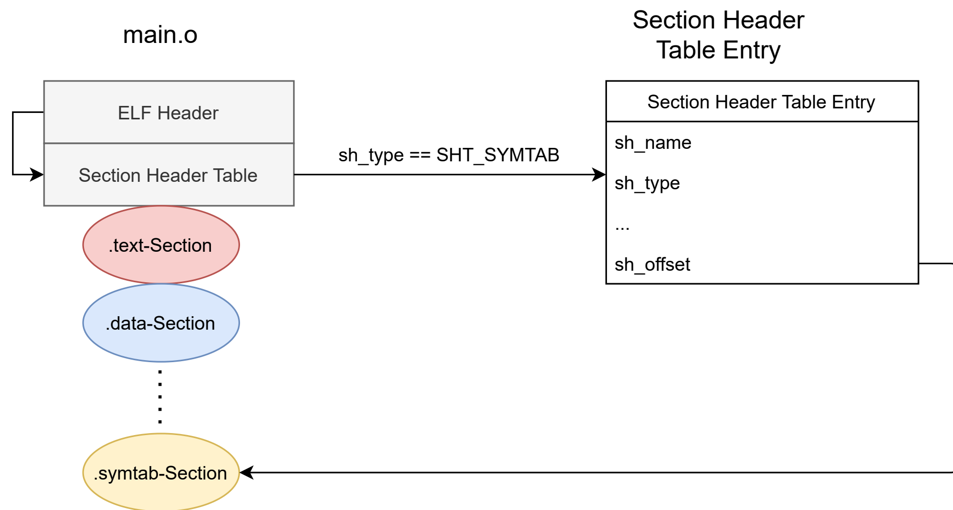


Figure 2.4 – Visualization of a symbol table lookup. The *linker* can parse the ELF Header to find the SHT. Every entry within the SHT holds a type which identifies its content. The magic value of *SHT_SYMTAB* tells the linker that this section holds the symbol table. The attribute *sh_offset* holds the file offset within the binary where the symbol table is stored.

turn if the symbol is a *global symbol*. Those symbols are globally bindable and other modules are able and allowed to reference them. This procedure is clear for use-cases where one module *defines* and zero or more other modules *reference* these symbols. Problems arise when the linker finds two or more equal symbol definitions for a given data object. A symbol's equality to another symbol is decided by its name. Imagine *foo.c* and *bar.c* both defining a function called *sum*. When *main.c* calls *sum* the linker does not know to which of the functions implementations it should link to. In such a case the GNU linker *ld* aborts the linking process by default for example[13]. The developer, however, is able to change that behavior by linking with the option of *-z muldefs* which lets the linker link to the first definition of a duplicate symbol. If the reader recaps she will argue that this is not a real problem to all symbol definitions, but to those being defined **globally**. Local symbols are excluded because these are only bindable to their corresponding modules. That means that the linker will not take those definitions into consideration. This is an important fact, and it will rise up again in Section 3.1.3. **Multiple definitions of the same global symbols must not occur. Local symbols are excluded from that because they are only bindable to their own modules.**

2.2.2 Relocations

The compiler makes use of symbols to address the issue with data object *declarations* and their actual *definitions*. For declarations, it will create undefined symbols to tell the linker that there is a dependency to another object which has not yet been resolved. What is now missing is the information where exactly this resolution has to take place within the object file. Because the compiler is the entity placing the symbols it is also aware of the location at which it was unable to resolve the data object dependency. It can transport that information to the next entity, the linker, by placing *relocation entries* inside the ELF that identify the locations which have to be modified. The linker is in charge to apply relocations so that the final executable now contains valid addresses for all data references. Since symbols and relocations are placed for all types of references they

apply to general function calls as well as accesses to data objects such as global integer variables for example.

One might now ask if the target address is not known at compile-time what does the *compiler* place as an address at that position instead? The answer is: it depends. In general there are two types of relocations. One of them stores an *addend* right at that position at which the actual address should reside. The other one saves this addend as a part of the *relocation entry*, so it is not placed at the relocation position which is then filled with zeroes. The latter is the one used the most nowadays[2]. This addend is just a value added to the target value to form the final address. This is necessary if we access the nth element of an array for example.

When a function is defined it can be called an arbitrary amount of times within an application. Therefore it is possible to have multiple relocation entries all referencing the same symbol, namely the function to resolve, as shown in Figure 2.5. The linker will find the undefined symbol, grab its relocations and updates all of their locations inside the final ELF by placing the valid function address. Just like the symbol table relocations are aggregated in sections as well. In essence *symbols* and *relocations* together enable the dependency resolution across multiple different objects files. Relocations describe the positions at which addresses have to be altered and symbols are used to identify and calculate these addresses.

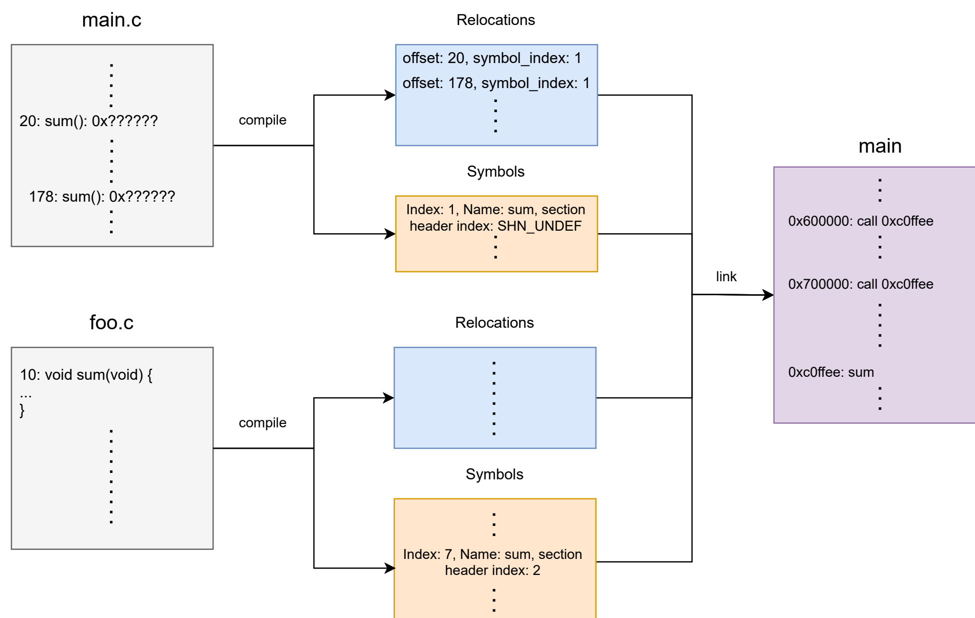


Figure 2.5 – Symbols and relocations. Together they allow dependency resolution by the linker. The relocations in *main.c* reference an undefined symbol *sum*. This symbol is defined in *foo.c* and can therefore be resolved during linking where the *linker* assigns addresses to *data objects*. It assigns the address *0xc0ffee* to the data object *sum* and can now relocate the calls of *main.c* at file offset 20 and 178. After relocation the calls to *sum* at addresses 0x60000 and 0x70000 succeed.

2.2.3 Memory overlay

Memory overlaying describes a technique in which multiple parts of an application can be placed at the same memory location, thus they overlap or *overlay*[1]. The memory overlaying is implemented at the linker level because it is necessary to build two or more parts of the main application image which reside at the same location. Since the linker assembles the whole executable and chooses feasible addresses for all parts of it, it's the entity which builds the overlay. Overlaying data results in several challenges to face since the overlaying parts reside at the same memory region and therefore references between these parts and the other data of the executable must remain valid for all overlaid parts as Figure 2.6 illustrates.

In order to model the shown behavior, the linker will choose appropriate addresses during the linking when it resolves relocations for example. For relocations of *.mem-overlay2* it will calculate relative addresses according to the same base address as *.mem-overlay1* because both of these sections share that address. The outcome for both of these sections is a situation in which they would be the only entity being loaded at their specific address. In fact only one part of any memory overlay unit can be *alive* e.g. present at any given time. The reader may notice that as an obvious fact since there is one memory location for n memory overlaid application parts. Questions arise about how this overlaying is implemented at runtime if only one overlay part can be present. **The concept of memory overlaying requires not only linktime but also runtime support.** A second entity has to apply the actual overlaying mechanism when the target application is in execution. This entity has to take the memory overlaying unit which is currently placed within the address space of the application and replace it with another unit - at that **exact** same memory location. This requirement directly translates into several constraints and one of the obvious ones is that all memory overlaying units have to be of equal size. In case of real application code it does not require these units to have the same amount of functions for example. It is just a necessity that the memory regions are of equal size, so one part could be smaller than the other, but then it has to include additional padding.

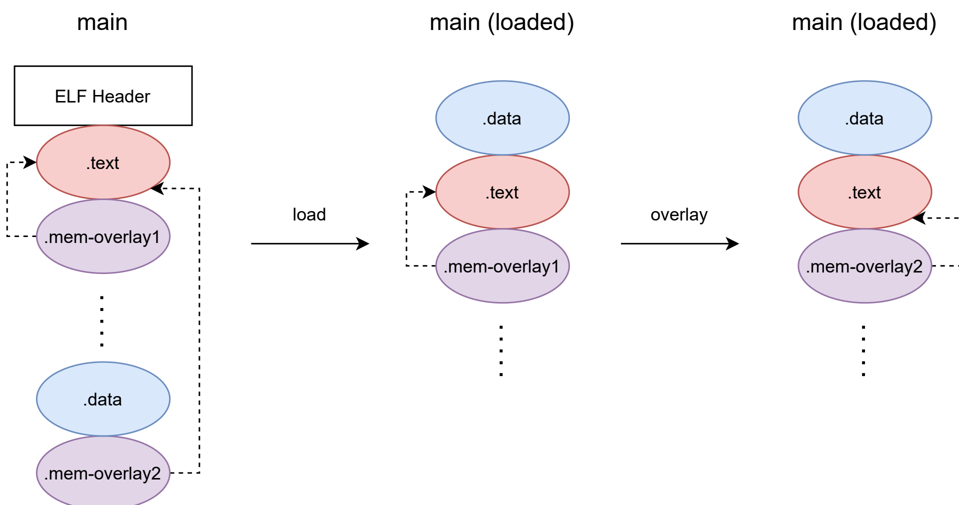


Figure 2.6 – Memory overlaying concept. The area to overlap here is described as the sections *.mem-overlay1* and *.mem-overlay2*. During the runtime of the application *.mem-overlay1* is loaded. At some point during execution *.mem-overlay2* is laid over *.mem-overlay1* thus replacing it. References of *.mem-overlay1* and *.mem-overlay2* into *.text* must remain valid at each moment of execution.

In essence memory overlaying is a technique which allows multiple parts of an application to reside in the same memory region one at a time. Linktime support is necessary to prepare and save the memory overlay units within the executable and runtime support is necessary to apply the actual memory overlay.

2.3 Address-Space Views

Memory overlaying describes one technique used to facilitate multivariant applications by replacing certain parts of the program during execution. Another approach is to not replace application data but to establish multiple views of the application which contain these different parts of the executable. With different views we transition from replacing specific aspects to switching between views to leverage multivariance.

The view of an application is the application's layout while it is loaded into the main memory in form of a *process*. The process model describes the application at runtime by properties such as its virtual address space, opened files, signals, handles and so on. A *thread* is the virtual execution unit which lives inside that process and is described by its internal state such as CPU registers or its stack. One can say that processes really are an abstraction of the whole computing system since they contain every resource a program needs to successfully execute and threads are abstracting the CPU itself. Since the virtual address space is a property of a process and threads live inside a process they commonly share the same virtual address space. To introduce multiple application views the process has to not include one, but multiple virtual address spaces.

Multiple ASVs loosen the tight coupling between threads of the same process by allowing different threads to live in different address spaces while remaining in the same process. Therefore, even though two threads belong to the same process they may have a different *view* of the process, or the application, in which they are running in. Because these two threads *see* two different versions of the same program this concept describes an application multivariance at the thread level. As one use-case Rommel et al. give a practical example for applying binary patches to different applications by creating a second ASV which contains the patched application version and to which each thread can individually switch to[15].

Figure 2.7 shows that the technical implementation behind ASVs is a combination of shared memory with individual Copy-on-write (COW) mapping for specific application parts. While most of the view of the program is shared only some developer-specified parts are different and diverging from each other. The reader can think of an example in where one view of the application logs a function call to the console while another view excludes this logging. When a thread switches its view then most of the application data remains the same except for the fact that this single thread will not log its function call anymore.

In essence Address-Space Views allow *threads* of the same *process* to have a different *view*, to see the program from a different *perspective*, and enable multivariance at the thread-level.

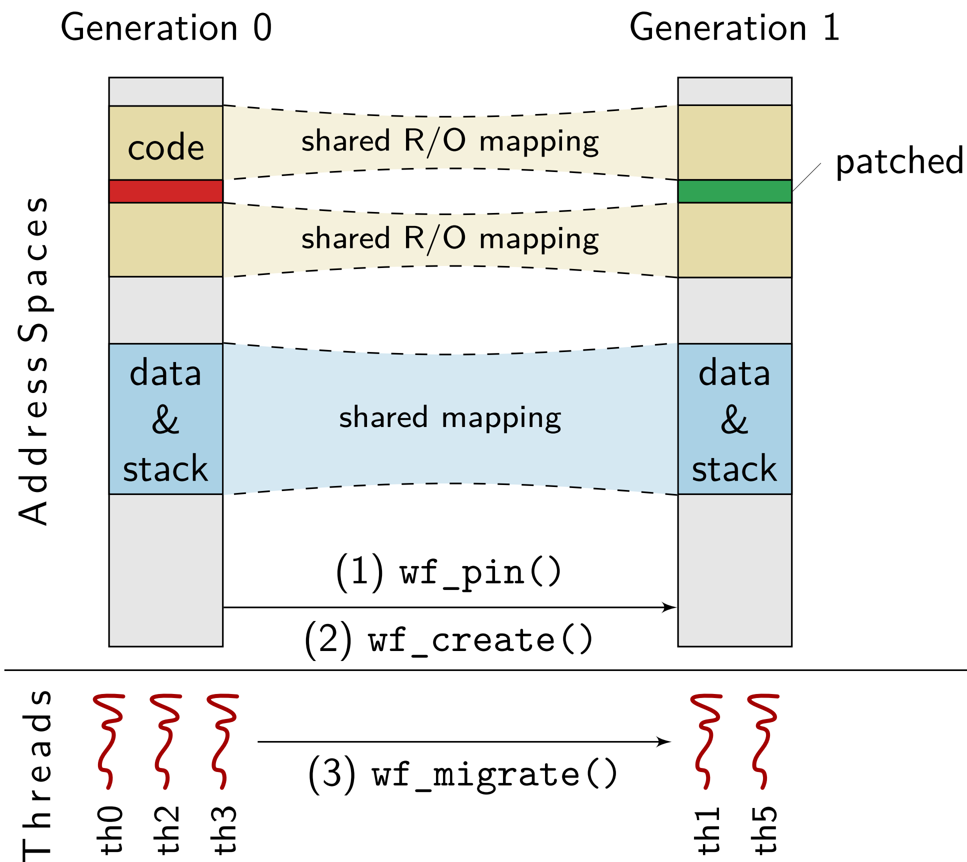


Figure 2.7 – Two ASVs next to each other. Most of the data between these two views is shared, but certain parts can be *pinned* which results in an own private COW memory mapping. The pinned part is marked in green. Figure taken out of [15, p. 5].

2.4 Related work

Multivariance in computer applications has been a topic of research for decades and this section will describe similarities between the thesis and this research as well as how it differs from existing approaches.

Dynamic linking provides a global layer of multivariance in application software by allowing programs to link against shared object files. The references to shared objects are resolved during the runtime of the application with the help of a dynamic linker and is already part of the Multics operating system since 1967[3]. Replacing such a file with another version of itself results in all applications using the new variant, thus globally applying multivariance. Robert C. Daley et al. describe the possibility of sharing procedures and other data as two desired properties of multi-process computer systems[3, p. 4]. The difference between that approach and the thesis is the area of effect of applying multivariance. While the replacement of shared objects has global impact to all applications of the computing system linking to that library this thesis aims at only incorporating multiple different variants of its own code. Additionally, all variants of an application are known at

linktime and embedded into the executable for the purpose of this thesis whereas dynamic libraries are separate files.

With the introduction of loadable kernel modules another layer of multivariance has been established within the operating system interface. These kernel modules allow a supervised user to extend the kernel's functionality by loading dedicated modules into the kernel's address space to add support for different types of file systems or network devices for example[7]. To leverage multivariance inside the kernel the user can first unload and then reload another variant of a specific module. The visibility of that modification is again global and every application which access the kernel module in any way will make use of the updated version. Following dynamic libraries, additional variants of loadable kernel modules are separate files and not embedded into the original module or the kernel itself. Additionally, they require a privileged user whereas the thesis' approach can be applied by any user which is able to link the application.

The paper of Rommel et al. introduced in Section 2.3 exploits ASVs to apply wait-free binary patches in multithreaded applications [16]. The authors elaborate on the latency impact of these binary patches induced by points of *global quiescence*. Global quiescence describes a situation in which all threads of a process have to halt at a globally defined gate within the application in order to make sure the applied patches do not corrupt the executing threads. This could be the case for example if the binary patch replaces program code while one or more threads are currently executing this program code which can potentially destroy the individual call stacks [16, p. 2]. Waiting for all threads at a pre-defined point of execution increases latency because some threads can reach that point (far) earlier than others, thus they block unnecessarily and cannot make any further progress. To circumvent that the authors transition from global to *local quiescence* in which each individual thread on their own decides when it is applicable to apply the desired patch. The technical implementation makes use of ASVs which contain the patched version of the binary and to which each individual thread switches to if it's feasible to do so. This binary patch expresses a different view of the executable which is why it is strongly related to this thesis. The difference between these two is, however, that the idea of this thesis is to create multiple views of the same executable right upfront - at compilation/link time - to facilitate multivariance. Every possible variant of the binary executable is known right when it is build whereas the binary patches are variants of the application which are build and applied later on when it has been already released.

Multiverse follows a similar approach in which the authors create specific variations of functions at compile-time and allow the user to switch between these variants at runtime [17]. That means these functions are highly tailored to the needs of the developer which results in several benefits of functional and non-functional properties such as execution time or energy consumption. The technical implementation of multiverse places these function variants within the binary just as the approach of this thesis creates an ELF with all its views, but there are several differences between them. First and foremost all parts of a multiversed binary are loaded at application startup. All functions and all of their variants are loaded right-away whereas this thesis does load only **one** view at a time and therefore the other views are not kept in the main memory. Secondly multiverse includes specific views of the application just as this thesis, but it is expressed at the granularity of **functions** and not at a more coarse-grained level such as a text-segment for example. Thirdly these views are applied differently in compared to this thesis. Multiverse patches all call targets, in other words all functions calls, which results in the application calling a different variant of the currently *active* variant of a specific function while the thesis overlays whole segments and does not patch specific control-flow paths. Lastly the applied multivariance of multiverse is visible to all threads at the process-level whereas this work makes use of thread-level specific multivariance by ASVs.

2.5 Summary

The reader has been introduced into the concepts being used within this thesis to accomplish multivariance in binary executables. He learned that the ELF is a file format for organizing and structuring executable files as well as object files or shared objects. ELF files are made up out of sections which aggregate related data in a compound manner. These sections have several properties assigned such as a name, size, file offset and many more. A section is part of a segment which generally describes a linear region of memory. The segments are the structural parts the loader processes and loads into the main memory. They do have several properties as well with access rights being one of them. All sections that are placed within the same segment share the same access rights.

The objects files which are organized as an ELF file are the outcome of the *compiler* which translates source code of a programming language into instructions the target architecture's execution engine, the Central Processing Unit (CPU), can understand. A compiler is not omniscient - it translates one source code file at a time which is why the objects files themselves are not executable. They only do describe a certain part of the main application and have to be linked together by the *linker*. The linker takes all object files of an application as an input and builds the final ELF executable. Source code files and therefore their object files can contain references between them, when one of them calls a function which is defined in another one for example. These references cannot be resolved by the compiler which is the reason it places *symbols* within the object files to transport that information. Next to symbols it also places *relocation* entries which describe the exact position within the object file at which addresses have to be altered in order make function calls or data access operations actually succeed. The linker will take those symbols and relocations and process them on all object files. It fills relocations with their valid addresses because the linker has *all* object files as an input and is therefore able to calculate them appropriately.

Since the linker calculates the addresses it is the entity to facilitate *memory overlaying* at first place. For memory overlaying the linker has to lay out several parts of the application to reside at the same virtual memory addresses, hence the wording overlay. Only one version of an overlay can be active at any time of execution which is why memory overlaying also requires runtime support in order to actually apply the overlay. Such an overlay manager takes the active overlay unit and replaces it with another overlay unit. Because the whole unit is overlaid one constraint to memory overlaying is that the overlaying units must be of the same size.

Finally to facilitate the overlay mechanism one possible technical implementation is the use of *Address-Space Views*. Address-Space Views are virtual memory mappings which have most of their data shared across all view instances while some specific parts are diverging. This allows to explicitly mark the aforementioned units, so that two views share all of their data except for the overlaid parts. With that approach a thread can switch to another view which results in that thread having the exact same view of the executable as before except for the marked overlaying part, thus leveraging multivariance by dedicated ASVs.

3 ARCHITECTURE

In this chapter the reader will be introduced into the technical and non-technical concepts which have been worked out over the course of the thesis' project. It will show and reason the thought process behind the approaches taken to facilitate multivariant applications. The explanation will start at the linker where the multivariant executable is assembled and continues to explain the development of a runtime environment to apply multivariance during execution. At the end a summary composes elementary implementation details and recapitulates key aspects of the developed architecture.

3.1 Expanding the LLVM linker

In the Chapter 2 the linker has been introduced as the one entity building up the final application by taking all of their modules, the *object files*, and assembling the executable by resolving cross-references between these modules. Therefore, the linker is an essential part in evolving application multivariance since it decides which parts of an application are included in the final executable. The *LLVM linker* has been the linker chosen to be enhanced over the course of this project because it is open source, well documented and easy to expand.

To facilitate multivariance ELF files have been augmented to include all user-defined *views* of the application within their binaries. To this time a view is only composed out of functions so other data objects such as global variables are not part of an application view. It is the first step of combining the benefits of *static variability* with those of *dynamic variability* because each of these views is a specific tailored *variant* of the application to which it can switch to dynamically during execution. The granularity of these views is on the level of *object files* which is why multiple views of the program are expressed by multiple multivariant object files. So the first step to manage multivariance within the application is to process multivariant object files.

3.1.1 Multivariant object files

Multivariant object files are object files which contain definitions for the same data objects. Imagine a scenario in which an application defines a helper library to ease database calls. The library is defined in *db-helper.c* and includes functions to read from and write to a database. When this library is translated the *compiler* takes the source code file and creates an object file for that code. To make use of static variability the source code file contains C preprocessor (CPP) expressions which regulate the translated source code by providing conditional compilation [4]. With the CPP developers are allowed to write code which is included in the final executable if a certain statically defined condition is met. Listing 3.1 gives an example of the use of the CPP.

```c
#define LOG

User db_get_user(int id){

    #ifdef LOG
    printf("Get user for id %d\n", id);
    #endif

    // Rest of logic.
}
```

Listing 3.1 – Usage of the C preprocessor. Program code between *#ifdef...#endif* regions are translated if the given name is defined. In this example the printf function call will be translated because LOG is defined in line 1.

The CPP is the concept leveraged within this thesis to build multivariant object files. When the developer decides to create two *views* of the function *db_get_user* then she would define the name *LOG*, compile it and then remove its definition and compile it again. The output is two object files, both with a function definition of *db_get_user* and one of these definitions would include an additional *printf* to verbose output information to the user. If the developer now tries to link all objects files to assemble the final application he would generally receive an error by the linker telling him that multiple definitions of the name *db_get_user* exist. This is a circumstance which has been explained in Chapter 2 already and one solution for the GNU linker *ld* has been by using the linker option *-z muldefs*. The *lld* allows the same option but the huge problem with that approach is that this option allows far more than the developer originally intends to do. The goal is to allow multiple definitions of data objects for all multivariant object files and not for all object files in general. The big difference is that multiple definitions of the same name across multiple object files can indicate an error within the application e.g. the developer mistakenly defined a function twice. The existing linker option levers out that guard and thus is the wrong option to choose for the multivariant approach. Therefore, the first extension of this thesis to the lld linker are two additional linker options which allows to enable and list all multivariant object files. For the purpose of this project the developer does also have to yield a number to all multivariant object files. The reason for that is that the final application binary will include multiple views of itself and one of these views has to be the *default view*. The default view is that view that gets loaded right at first application startup while the other views remain in the application image on the disk. All in all the established linker option takes a digit and an object file name and is usable an arbitrary amount of times to list all multivariant object files.

The example in Listing 3.2 shows how the new linker options can be used. The declaration order of the multivariant object files is irrelevant for the choice of a default variant since the linker option *mvo* with the value 0 indicates the default view. Within the lld source code tree that linker option has been added to the option list and the existing argument parser has been altered. Its code base

```
    ld.lld main.o foo2.o foo1.o --multivariant-linking --mvo 0:foo1.o --mvo ↘
        1:foo2.o
```

Listing 3.2 – Example code for linking with the new linker options.

already includes helper methods to parse argument lists such as *mvo* which is why the only thing that has been done was to create a new *vector* to hold the given information. The data is saved as a vector of pairs of *int* and *string* to collect all object files of all views. With such a vector the developer is able to define an arbitrary amount of object files to be part of a view identified by its preceding view index.

Now that the developer listed all multivariant object files the lld has to allow multiple *symbol definitions* for the specified object files. This is necessary because the granularity level of the multivariant approach is at object file level and that means the linker will see multiple data object definitions of the same object. But that is exactly what the developer intends to do - to create multiple views with each of them having their own implementation. Because of that the symbol resolving has been modified to enable that mechanism. Symbol resolve is a global process in which the linker gathers all symbols out of all object files to connect *symbol declarations* to *symbol definitions*. It is also that part which throws up originally if multiple symbol definitions of the same symbol take place. Because the goal of this thesis at first was to *enable* multivariance in applications by augmenting the ELF the resolving is implemented straightforward. **When multivariant symbol resolve is executed the last symbol definition is taken.** That means that the linking order of object files is highly relevant to the symbol definition taken, but it only applies to data objects not being functions such as global integer variables. For functions the linker will use the definition out of the default object file.

With these two actions, creating new linker options and allowing multiple symbol definitions for multivariant object files, the developer is able to successfully link the files to create an executable. The following use-case exemplifies the effect of the established symbol resolve process and is displayed in Figure 3.1: Imagine the example function *db_get_user* is getting *declared* in the main source code file and a function *get_user* out of main calls *db_get_user*. The developer linked two files with both **defining** *db_get_user* in *foo.c* but made sure that the object file *foo2.o* has a 0 appended as its preceding view index. Applying the mentioned approach results in the linker resolving all references to *db_get_user* using that object file which is being listed with index 0 as their *--mvo* linking attribute. For the given scenario that means that even though *foo1.o* could be linked "first", e.g. being listed before *foo2.o* in the object file link list, the function definition of *foo2.o* will be taken to resolve the call in main and all other calls to that function. For now the definition in *foo1.o* is left untouched and will never be called by any program code during execution. Therefore, in that scenario *foo2.o* acts as the **default view** of the application and *foo1.o* is an **alternative view**. The default view is that view, that *perspective* of the application, that is run per default during program startup. An alternative view is a view to which the developer can switch to during the runtime of the application process.

To enable that switching a runtime has to know *what* has to be switched and *where* it can find it. For that reason the *lld* has been amplified to record every function of every multivariant object file. With that information it is able to replace or *overlay* one function of a view with another function out of another view. In order to find these functions the *compiler* is called with an argument that creates an own section for each and every function definition in one source code file. The only thing the lld has to do is to gather all of these sections and save them in some sort of *view containers*. A view container is a container holding all function definitions of a specific view so that the lld can distinguish between them across multiple multivariant object files. In this project these containers are two-dimensional vectors where an index into the first vector selects the view and another index into that view selects the function definition.

When the linker iterated over all object files and collected all input file sections it has collected all multivariant function sections as well. At this point it is able to wrap all function sections of a multivariant object file into one output section which is then written to the output file. That means that for every multivariant object file the linker will now create one output section *.gen* to

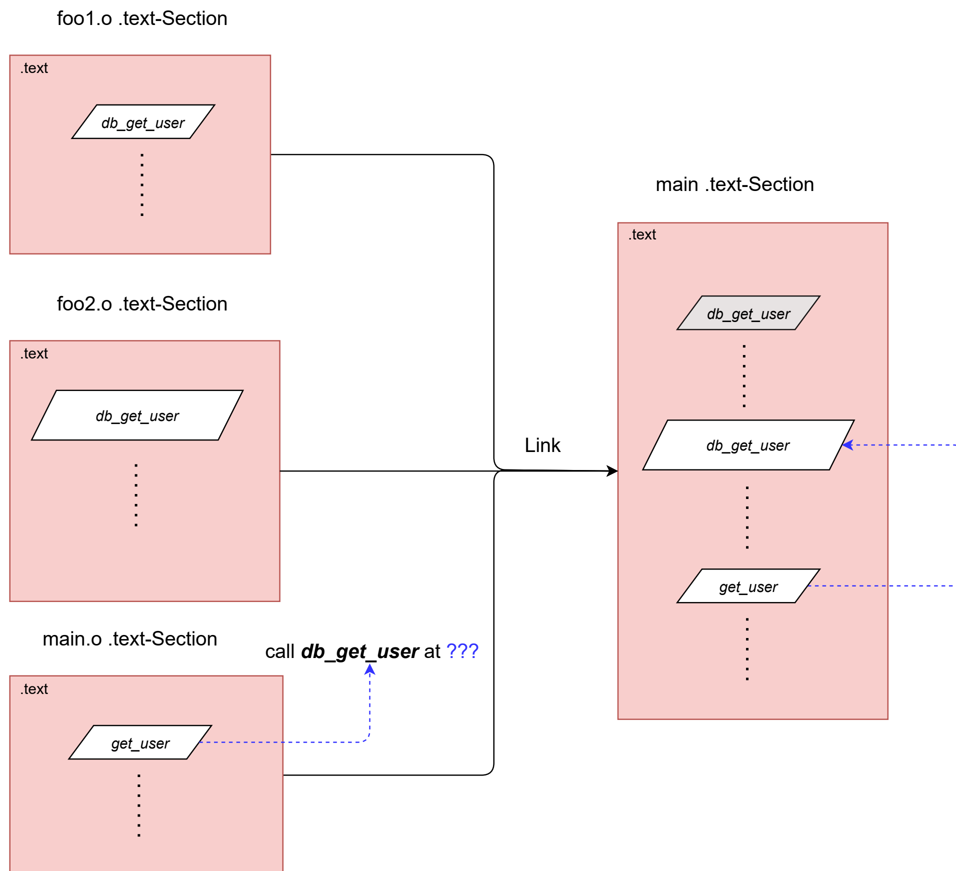


Figure 3.1 – Symbol resolve for multivariant object files. The function symbol *db_get_user* is defined in *foo1.o* and *foo2.o* and declared in *main.o*. *db_get_user* is bigger in *foo2.o* because it contains logging logic which *foo1.o* does not. The linker argument call lists the statement *--mvo 0:foo2.o --mvo 1:foo1.o*. When the linker resolves that dependency in a multivariant scenario it takes the definition that is given as index 0 in the linker argument list for the *--mvo* linker argument. The result is all declarations of *db_get_user* are connected to the definition within *foo2.o*. The definition within *foo1.o* is included in the executable, but never called.

distinguish between the different views of the executable and Figure 3.2 gives an overview of their layout within the final binary. These sections are paged-aligned within the application ELF which allows for efficient memory mapping of that file as Section 3.2 will further describe.

With all multivariant functions being collected the linker can solve the problem of equally-sized overlay units as described in Section 2.2.3. When units are overlaid is it essential that they are of the same size because only whole units can be overlaid. That means that all variants of a function must be as big as its biggest variant. Figure 3.2 also visualizes the usage of additional padding for all other variants. When the linker determines the output section size of a function section it now checks if that function section is part of a multivariant object file. If this is the case then it does not use the real section size of that function but rather the size of the biggest function variant. It allocates enough space in the output file to fit the function plus the padding bytes if there are any. Applying that procedure to all multivariant function definitions ensures a consistent equal size of all memory overlay units.

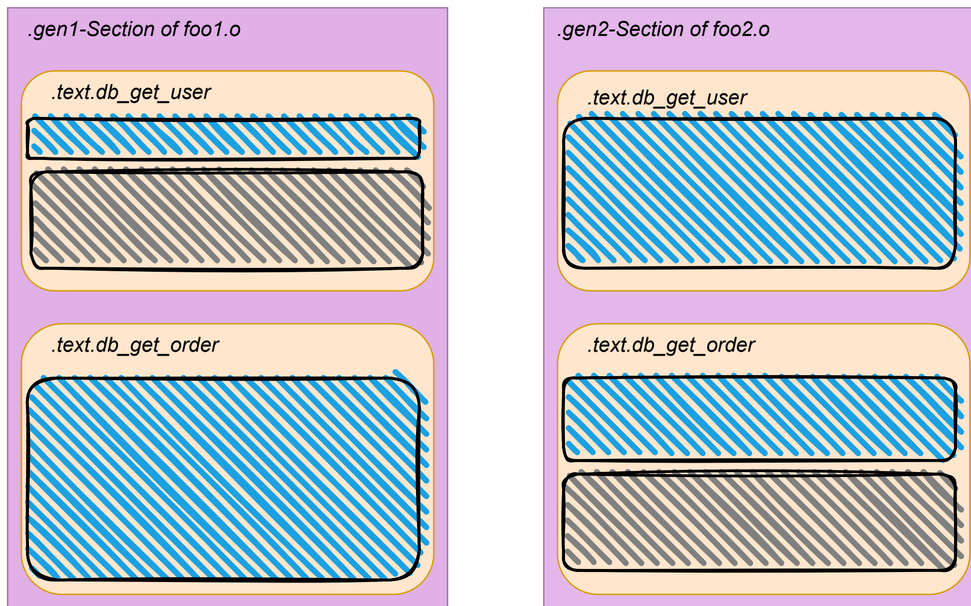


Figure 3.2 – Comparison of two function variants of *db_get_user* and *db_get_order*. Code is drawn as blue-shaded area, padding is grey-shaded. These functions are part of a memory-overlayed unit, so they must have the same size. The smaller function definitions contain padding bytes to reach the same size as their bigger counterparts in the other object file.

3.1.2 Deploying memory overlay

To deploy memory overlay units they have to be formed at first place. Within this project the granularity of multivariance is statically expressed at the level of objects files. During runtime, however, it makes use of *segments*. Since segments only describe are continuous space of memory one segment can overlay another one by being loaded at the exact same base address and having the same size and access rights. The idea is to overlay all functions of the running view with all of their counterparts of a loadable view in a way that they perfectly lie on top of each other. Therefore, the linker does create additional segments with one segment for each view of the application. Each memory overlay segment contains all functions of all multivariant object files of the same view index so that the switch from one view to another results in all function variants being switched to the new view. Since the requirement for the overlaid units is that they are placed at the exact same addresses the linker has further been modified to respect these multivariant segments. When the linker now processes an output section which is part of a multivariant segment it ensures to assign the same base address to these output sections for all variants. That means that two function sections of the same function out of two different object files will get the same virtual address.

In further assembling all references to these sections are calculated in respect to their same base virtual memory address, thus facilitating memory overlaying. This base address has to be page-aligned as well which is a direct constraint out of the usage of Address-Space Views as a runtime solution to memory overlaying. Section 3.2 further explains that the pinned memory regions which are allowed to diverge are set at page boundaries.

Another important part is the lifetime of these multivariant segments. Section 2.2.3 explained that only one view can be active at any given execution time, which means that all other views are unused and that it is not necessary for them to reside in memory when they are inactive. Segments

have a type assigned which describe if they should be loaded or not and this type is used to describe additional views apart from the default one. The linker will mark exactly one multivariant segment as loadable segment while all other segments of all other views are marked as not loadable. The result is an application in which the one default view segment is initially loaded while all others remain on the disk as Figure 3.3 illustrates.

When the views are switched during runtime what actually changes is function implementations of one view are replaced by implementations of another view. What is not switched is the *data* that the view uses because it is not part of the memory overlay unit. Even though each view has its own data part which is part of its multivariant object file that data cannot be switched. **All application views refer to the same data.** That means the current implementation of this work does not support *view-local* data. All views must define the same data, even if that means that a view defines data which it doesn't use by itself but which is used by another view. Simply spoken the set of data for every view is *total* since it contains every possible data item across all existing views. Section 5.3.2 will discuss the benefits and drawbacks of this approach whereas Section 5.5 will give a suggestion of how the *total* data sets can be altered in future work.

The previously mentioned *symbol resolve* does apply to all kind of data objects and therefore to the global data set of all views as well. This mechanism on its own already ensures that views implicitly access the same global data. Because always the last definition of such a data object is used for resolving all view instances implicitly access the same objects, thus they are synchronized in data access. It does not, however, cover symbols for local data such as those being declared as *static*. Local symbols take a different path within the lld and enforce a separate processing. All views must access the same local data across all view instances just as they do for global data, so local symbol processing has been modified within this project as well.

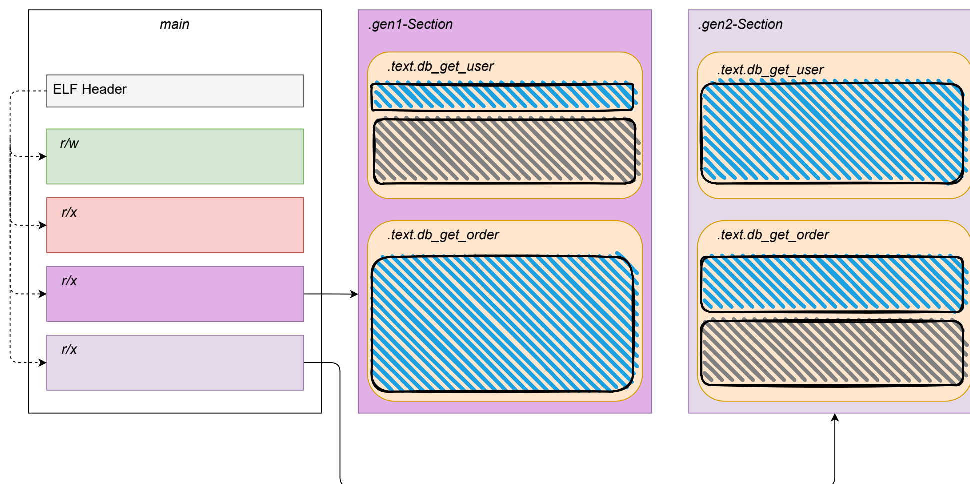


Figure 3.3 – Multivariant segments and output sections in the ELF file. The main executable has two additional read/execute segments colored in purple. The first purple segment behaves as the default segment, it is loaded initially during application startup. Both segments contain an output section *.gen* which bundles all input function sections of a multivariant object file. During runtime the first segment can be overlaid by the second segment to switch from one application view to another.

3.1.3 Local Symbols and Relocations

The processing of local symbols differs from the one of global symbols because they have a different semantic meaning. While it is a race condition when two equal global symbol definitions exist in two different object files this is not an issue for local symbols. This is due to the fact that this symbol is only visible to the module defining it, so the linker will not take it into consideration when resolving symbol references across object files. The solution for global symbols has been to replace equal global symbols so that the last definition of a symbol is taken. Since the lld does not compare local symbol definitions across object files it has been enhanced to keep track of all local symbols of all multivariant object files. When relocations are done for local symbols the lld checks if the requested symbol definition stems from a multivariant object file. If that is the case it will **always** return the symbol definition out of that file that has the index 0 for the --*mvo* argument list. Because a constraint of the implemented architecture is that all data sets of all views are *total* it is ensured that the processed symbol is defined in every multivariant object file. The linker affords that by returning the proper *data section* of the local symbol when calculating its virtual address which is the section of the aforementioned object file with view index 0. Returning always the same data section for each local symbol definition across all multivariant object files leads to the same relocations for all views and the result are consistent, synchronized data accesses to the same data object across all views. Figure 3.4 illustrates the impact of that approach on the relocations of these object files.

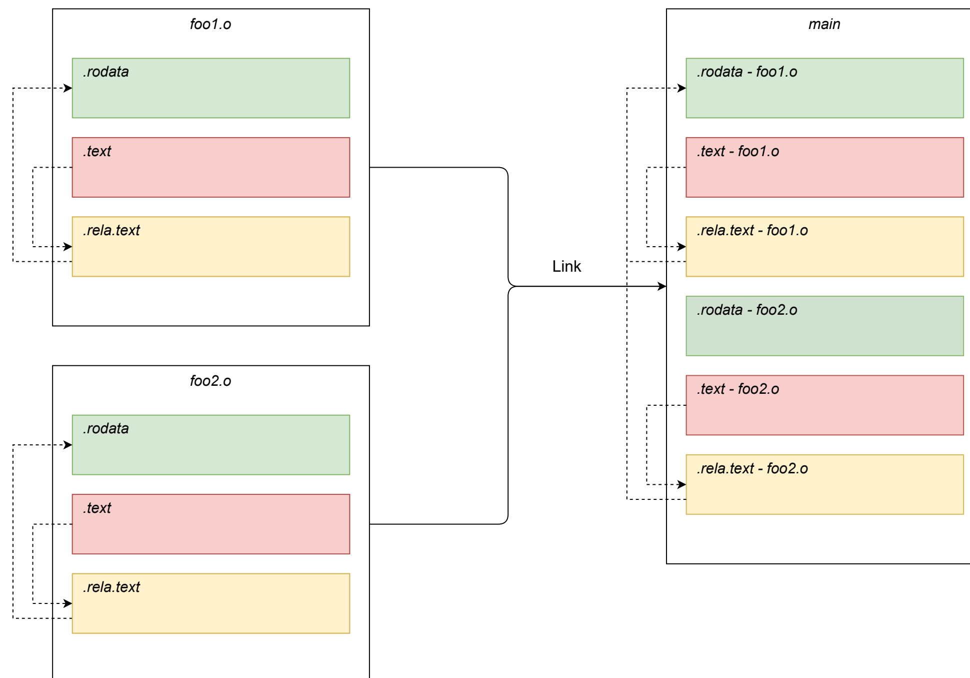


Figure 3.4 – Relocations of local symbols of two multivariant object files *foo1.o* and *foo2.o*. Each object file contains a *.rela.text* section which contains all relocations for the *.text* section. The references of these relocations are towards the local symbols of their own *.rodata* section prior linking. During linking the linker redirects these references to the symbol definitions of *foo1.o* because its view index has been 0. In the end that leads to *.text - foo2.o* accessing not its own data but the data of *foo1.o* so that both views refer to the same data objects.

Applying these modifications to the lld linker allows to place all necessary information in the final ELF binary for a runtime to overlay different views of the application during execution. To this point, however, the runtime is not able to find this information and therefore a protocol has been established within this project to pass that information to a runtime.

3.1.4 Preparing data for the runtime

In order to allow a runtime to find view information the linker embeds additional metadata into the application ELF describing the view contents. This information is placed in an own section *.genmeta* as a structure and includes the following information:

- Size of the structure describing this metadata in bytes
- Base virtual memory address for all views
- Size of a view in bytes
- Number of views embedded in the ELF
- Array of file offsets of all views inside the ELF

The meta section does not need to be placed in an own segment, it is appended to an existing read-only segment and loaded initially at application startup. To be able to find the meta section itself the linker places an additional symbol *genmeta_start* which contains the virtual address at which this section can be found. This symbol is exported as a dynamic symbol which allows the runtime to find that symbol when it parses the ELF.

While the creation of that meta section is the next step after parsing all input file sections of all object files filling its contents is deferred to a later point of linking. The reason for that is that after parsing all input sections the linker has not yet decided at which file offsets these input sections will be placed in the output file. The lld, however, expects all input sections which should be placed in the final binary to be defined at a very early stage during linking. Without deeply altering the linking process of the lld it is not possible to create that meta section at that point of time at which file offsets have been applied to all sections. Therefore, a workaround has been implemented to solve that issue by creating a *dummy* meta section right after all input sections have been parsed. Even though the exact file offsets of the views within the ELF is unknown at that time the linker still does know how many views will exist because it parsed all input sections. That means it is capable of allocating an empty meta section with enough space to fit in that data when it is ready. It calculates the size needed to save the metadata structure and creates its section without actually placing the metadata at first place. Later on, directly after all file offsets have been applied, the meta section is filled with the listed data above.

Figure 3.5 gives the reader an overview of the final multivariant ELF executable and summarizes how it is structured. It contains all views of the application to which the process can switch to during runtime as well as the metadata to identify those. A proper runtime environment can parse that ELF and apply the view exchange mechanism.

3.2 Incorporating a runtime

Building an ELF which utilizes memory overlaying to facilitate multivariance requires a runtime overlay manager to apply that overlay. Within this thesis a runtime library has been developed

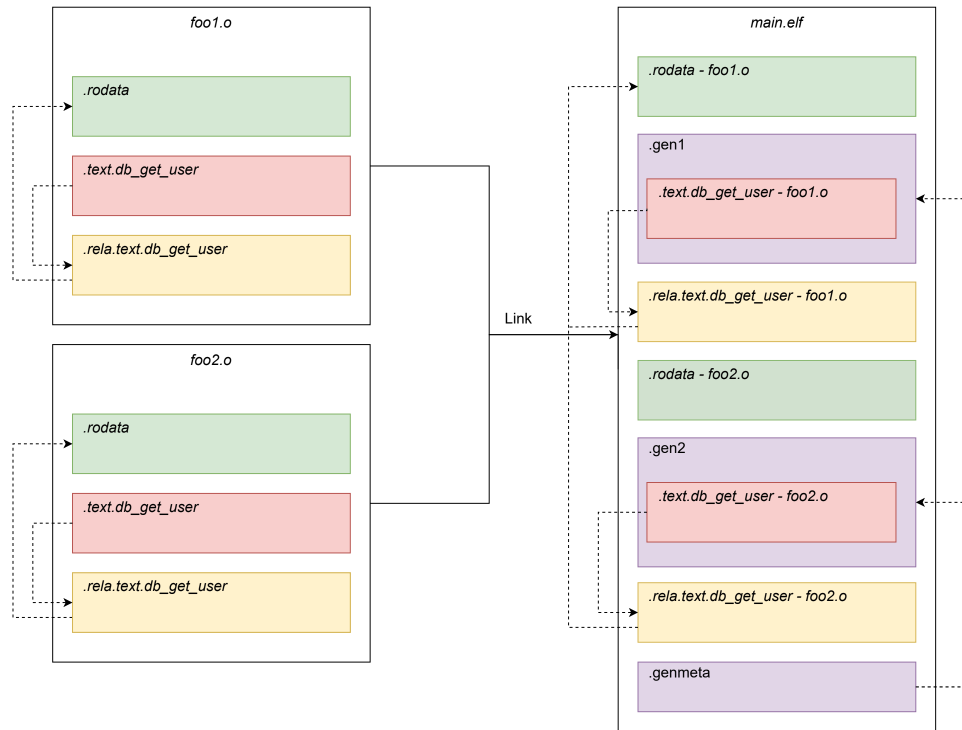


Figure 3.5 – Final layout of a multivariant ELF. *foo1.o* and *foo2.o* are multivariant object files which both contain a function definition for *db_get_user*. Before linking the relocations for *db_get_user* are directing towards the object file's individual read-only data sections. For the final executable the linker created two new sections, *.gen1* and *.gen2*, which both express two views within the application. Each of them contains a variant of the function and the linker made sure that both views access the same data objects by manipulating the relocation entries of *foo2.o*. The *.genmeta* section contains the meta information necessary to identify the multivariant sections during runtime. The sections of *main.o* have been omitted due to lack of space.

which parses the processes' binary file and allows the developer to switch between the views during execution. The library is built as a dynamic library to which the developer can link to and which exports the method *load_and_switch_view*.

First the method tries to load the metadata out of the processes' memory. It does that by calling the method *dlsym* which returns the address of a dynamically exported symbol. This is the reason why in Section 3.1.4 the linker decided to export the symbol as dynamic symbol so that *dlsym* can find it. When the address is found it statically casts that portion of memory into the structure defined in Section 3.1.4 to retrieve the metadata.

After the data retrieval the library starts to *pin* the memory region of the memory overlay unit which starts at the virtual address designated by the meta information. Pinning a memory region is the first step in utilizing ASVs, and it marks the area of memory which is allowed to diverge across multiple views. The function *wf_kernel_pin(start, end)* takes a start and an end address whereas the end address is calculated by the start address of the memory overlay unit plus its size. When the pinning succeeded the library calls *wf_kernel_as_new()* which creates the new ASV. The return value is the view id, and it is used to directly switch to that view by calling *wf_kernel_as_switch(id)*.

When the switching is done the thread currently executing *load_and_switch_view* is inside the new address space and starts to memory map the ELF binary of its process executable. Under Linux the file under */proc/self/exe* contains the path of the current executable which the library can *open* and *mmap*. *mmap* allows to memory map a file into the processes' address space. It further allows to pass an offset into the file described by the file descriptor of *open* so that the library is able to pass the file offset of the requested view. Since the offset must be a multiple of a memory page size the linker aligns the *.gen* sections accordingly [14]. With the file being mapped the library will execute a *memcpy* to replace the existing version of a *.gen* section by a new version of *.gen* out of the ELF binary. This is the core process of the memory overlay manager, after the *memcpy* the active view has been replaced and the new view is ready to execute. To ensure that page access bits allow these types of modifications two *mprotect* system calls mark the view area as *writable* before and as *read/execute* after the replacement.

Now the library only has to clean up by unmapping the ELF out of the process address space and by returning the ASV's index in which the executing thread now resides in. Since the loading and switching of different views is only necessary if the current thread is not inside the requested view already the library includes a thread local variable *activeIndex* which describes the threads view index. It has to be stored within the Thread Local Storage (TLS) because each thread can reside in its own view and TLS allows to model that behavior on a per-thread basis. If a second thread calls *load_and_switch_view* after a first thread already set up the requested view all the mmap'ing and memcpy'ing is skipped instead and the thread simply switches to that view by only calling *wf_kernel_as_switch(id)* and setting its private *activeIndex*.

At a later point in time a switched thread can decide to switch to the default view again by calling *wf_kernel_as_switch(0)* because the view with index 0 is always the default view.

3.3 Summary

In this chapter the reader learned about the technical and non-technical details of the architecture developed during this thesis to implement multivariance with specially crafted ELF files. At first, she was introduced into the changes made to the LLVM linker *lld* to enable processing of multivariant object files. Duplicate symbol definitions are allowed and wanted for these object files so the lld will now accept those as well as makes sure that for every duplicate definition the last one is taken. Function definitions are an exception to that, for those the lld uses the definition of the default view with index 0 that the developer defines via the *--mvo* argument switch. After that it collects all multivariant function sections and places them into dedicated output sections *.gen*. A *.gen* section bundles all function sections of a view and the linker makes sure that equal function sections across all views are of the same size by applying padding where needed.

With all view output sections created the linker creates additional segments for each of these sections. All view segments reside at the same memory location and are of the same size, thus forming the first step in deploying memory overlay. One of these segments is declared as the *default segment* and is loaded initially during application startup while all other view segments remain unloaded.

Because all views come with their own data sections such as *.data* or *.rodata* the linker also makes sure that each view accesses the same local data object as well. It does that by manipulating relocation entries and therefore does not allow defining *view-local* data. The manipulation is done by returning the same data section for each and every possible local symbol in all multivariant object files. After that all relocations of all views refer to the same sections and access the same data.

To allow a runtime to find the view information a protocol has been established to transfer that information. A specific structure holds all meta information necessary to gather all view information and is placed within a *.genmeta* section. The linker adds a dynamically exported symbol *genmeta_-start* to allow retrieving the virtual address of that section during execution. The filling of the meta section is separated from its creation because output section file offsets are needed, but only available at a late time during linking. To circumvent that issue this thesis introduces a dummy meta section right at the beginning which acts as a placeholder. When file offsets are calculated that section is filled and the meta information is complete. At this point of time the final multivariant ELF is built.

Actual memory overlaying is provided by a dedicated runtime library to which the developer can dynamically link to. It offers a *load_and_switch_view* function to facilitate view exchange by overlaying the view parts. That functions pins the memory space in which the views reside which allows them to diverge across all view instances. It then creates a new and switches into that address space to memory map the processes' binary. When the executable is mapped the library *memcpy*'s the content of the requested view out of the mapped ELF into the memory area at which the currently active view is placed. This forms the core mechanism of memory overlaying established in this thesis. After that it cleans up by unmapping the ELF file mapping and returning the active view index to the user. A subsequent call to *wf_kernel_as_switch(0)* switches the executing thread back to its default variant.

In essence the specially crafted ELF binary file as well as the developed runtime library together allow the user to switch between different views of the application during execution.

4 EVALUATION

To measure the effects of the implementation several test cases have been designed during this thesis. Two synthetic test cases will evaluate if the built architecture can be applied to an application in general. After that *memcached* has been chosen as a real-world evaluation target to test the developed architecture in a software which is established and used for several years already.

4.1 Synthetic test cases

The main purpose of the synthetic test cases is to prove the pure functionality of the thesis' approach. They consist of two settings in which both, the augmented ELF alone as well as the ELF plus the runtime is tested. The evaluation target system is a Debian 10 running a *wfkernel* which is a kernel that adds the necessary system calls to its interface to be able to use the Address-Space Views.

The first test involves an application which outputs only a test message. The code is given in Listing 4.1 and makes use of the CPP to control the output behavior of the application. To translate the source code into an object file the *clang* C compiler has been invoked with the following arguments: *clang elf.c -c -D LOG -ffunction-sections -o elf.o*. This command builds the first multivariant object file and the *-D* argument switch allows to pass a macro definition, so that *LOG* is defined and the presented code actually outputs the defined text. The second build command almost looks the same but omits the *-D LOG* so that no text is printed and is saved as *elf2.o*.

```
1  #include <stdio.h>
2
3  const char *logMessage = "This thesis rockz!\n";
4
5  void thesisLog(void){
6      #ifdef LOG
7      printf(logMessage);
8      #endif
9  }
10
11 int main(){
12     thesisLog();
13     return 0;
14 }
```

Listing 4.1 – Synthetic test case example code.

After both multivariant object files are built the updated lld linker is executed to link the final ELF. The example command is shown in Listing 4.2 and leverages the two new arguments *--multivariant-linking* and *--mvo*. The executable links successfully and creates the binary *elf* which can be run on the evaluation system. Executing the application results in an output of *This thesis rockz!* and the program exits calmly. This test case proves that:

- A multivariant ELF can be built out of multivariant object files being compiled with *ffunction-sections*
- The ELF is able to run successfully on the evaluation target system
- The default view defined by *--mvo* is the view being initially executed

A subsequent altered command with the default view being changed to *elf2.o* instead of *elf.o* by assigning the 0 view-index to that object file results in no output, thus further proving that the linker chooses the executable's default view upon the developer's preference.

```
1   /path/to/ld.lld [...] elf2.o elf.o [...] -o elf --multivariant-linking ↘
        --mvo 0:elf.o --mvo 1:elf2.o
```

Listing 4.2 – Command to link the multivariant executable. The listing of important standard libraries to link against are omitted, the focus is on the new aspects of the linker interface.

With the newly crafted ELF being able to run the next step is to apply the actual memory overlaying by the runtime in order to switch between these two views during execution. To do that the example code has been modified in Listing 4.3 to first call the view switching and second place additional output to increase comprehensibility.

At the beginning the default variant is called, followed by a switch to the second view at index 1 in the executable. A subsequent call to the log method shows the effect of the memory overlay as

```
1   #include <stdio.h>
2
3   extern int load_and_switch_view(int);
4   const char *logMessage = "This thesis rockz!\n";
5
6   void thesisLog(void){
7       #ifdef LOG
8       printf(logMessage);
9       #endif
10  }
11
12  int main(){
13      printf("Execute default variant:\n");
14      thesisLog();
15      load_and_switch_view(1);
16      printf("Switched to view at index 1. Execute variant:\n");
17      thesisLog();
18      load_and_switch_view(0);
19      printf("Switched back to default. Execute variant:\n");
20      thesisLog();
21      return 0;
22  }
```

Listing 4.3 – Synthetic test case example code.

another call to *load_and_switch_view(0)* complements the test by bringing the application thread back to its initial view. With the last call to the log method the test is done and the total overlay mechanism has been evaluated. Listing 4.4 provides the exact command-line output and proves the correct functionality of the taken approach of this thesis. The binary has been linked as shown in Listing 4.2 with the default view being the one in which the log method actually prints out the message. While the first call to *logThesis()* prints the message, the second call, after switching the view, does not print any further characters. The succeeding text shows that the pinned area includes one virtual memory page starting at 0x203000 and is a direct output from the helper library which calls the *wfkernel* system calls. When the thread switches back to its original view the statement is printed again, thus the runtime managed to overlay the views parts successfully during execution.

```
1   Execute default variant:
2   This thesis rockz!
3   wf-userland: memory pin [0x203000:+0x1000]: rc=0
4   Switched to view at index 1. Execute variant:
5   Switched back to default. Execute variant:
6   This thesis rockz!
```

Listing 4.4 – Command-line output of the view switching test case.

With the synthetic test cases showing that the developed approach can be applied to a small test application the next step is testing how it applies to a real-world application.

4.2 Memcached as a real-world application

Memcached is a distributed memory object caching system which operates as an in-memory key-value store [5]. As a cache it reduces the latency for several data accesses by saving an in-memory copy. A common use-case for memcached is to reduce latency for database accesses since its file remains on the disk for many database types which is typically slower than the traditional Random Access Memory (RAM).

To evaluate the area of effect of this thesis several micro-benchmarks have been measured. When the memcached server is asked to return data for a specific key its internal *process_get_command()* iterates over a hash map to find the corresponding objects. The client can optionally send a multi-key request which includes more than one key at once and retrieves all objects matching the given keys. This method is the evaluation target for the micro-benchmarks, and it is measured in two different ways.

In the first version memcached uses an if-statement to configure key printing. When a client connection has been established memcached decides whether it should log all keys for this client connection and sets a global boolean variable. That variable is read during key iteration of the multi-key request and each key is printed with respect to the state of that variable. The default state of that global variable is *false* to **not log** the keys of the get command. If the client ip matches a specific pattern the global variable is set to *true* to log all keys.

The other version of memcached contains two views of memcached in its binary and applies the thesis' approach where one view prints all keys and the other view does not print any key. Once a client connects memcached decides whether it should change its view for that connection and by that either logs all keys or no key at all. Because of memcached's concept of worker thread pools the handling thread has to switch its view only once upon client connection start because memcached ensures that this thread will be the only thread operating on that connection until that connection is closed. To match the other version's default state the default view of memcached is that view that

does not log any key. If the client ip matches a specific pattern the thread will switch its view to log all keys of the request. For both versions the following evaluation strategy has been applied:

- Each get request contains the same amount of keys across both versions
- Four worker threads are used to handle client connections inside of memcached
- Each worker thread measures the execution times of 1,000,000 calls to *process_get_command*
- For one measure this procedure is executed five times in a row
- The measurement method is using `clock_gettime` and the `CLOCK_THREAD_CPUTIME_ID` clock id

All in all one measure includes the execution times of 20,000,000 calls to the micro-benchmarked function because four threads measure 1,000,000 function calls five times in a row. To investigate the effect of the specialized logging view of the second version of memcached three measurements have been gathered with a client issuing multi-key get requests containing 25 keys, resulting in a total of 60,000,000 measured function calls for each version of memcached. That means for every get request the first version has to evaluate the former amount of if-statements whereas the specialized logging view omits these evaluations. 25 keys have been chosen to represent a fairly high workload. The client application is single-threaded and started four times to create four processes, each creating a single connection to match the four worker threads of the memcached server.

The results are tabulated in Table 6.1 and visualized in Figure 6.1 to graphically compare both versions. In Table 6.2 and Figure 6.1 the measurement has been repeated, but this time the default state for both versions was to log the keys of the command so they both switch to a do-not-log state as the client ip matches. Both results indicate no significant positive impact of the developed approach on execution times of the micro-benchmarked function. Even though both versions differ in several nanoseconds Section 5.4 discusses the impact and reasons for these differences. The complete measurement has been repeated using the `CLOCK_MONOTONIC_RAW` clock id, but that did not provide any significant performance benefits as well.

After the micro-benchmark, which is measured at the memcached server, another measurement at the client has been executed to detect any latency impacts on the client side when applying the thesis' approach of multivariant executables. Furthermore, the evaluation scenario has been modified to not measure the effect of omitted print statements but to evaluate whether a specialized application of the *Address Sanitizer* inside of memcached can have positive impact on client latency. The idea of the Address Sanitizer is to provide the detection of several semantic programming errors such as use-after-free bugs or out-of-bounds accesses to stack and global variables[10]. A limitation is, however, that the developer can only apply that type of sanitization to the whole program or explicitly exclude certain parts of the application from sanitization. If she intends to only sanitize one function for example its current implementation does not allow to mark that function. One solution could be to exclude every other function inside the code base, but that can be tedious and impractical for huge projects. With the established approach the developer is able to build two views of memcached - one which does not sanitize anything and the other one sanitizing exactly what should be sanitized. Because the granularity within this thesis is at object-file level it is currently not possible to solely mark one function, but it can be used to ensure that only that object file will be part of address sanitization in total. To evaluate thesis' effect on client latency one version of memcached executes a full address sanitization while the other version makes use of the two described views. The measured procedure is the *memcached_get* function of the libmemcached helper library. That method takes the same 25 keys out of the first memcached evaluation and returns the corresponding

objects matching them. This time the client application spawns 32 additional threads with each thread starting its own connection to the memcached server. These threads continuously call the *memcached_get* function until the main thread told them to stop. Each thread has its own vector inside the TLS which stores the execution times for each function call. Again `clock_gettime` with `CLOCK_MONOTONIC_RAW` has been used to measure the function latency. After starting the client threads the main thread sleeps ten seconds to let the client threads execute function calls without measuring. These ten seconds are explicitly not measured to allow the network and all participating entities to set themselves up. This thesis expects to reach a rather continuous workload after the given ten seconds. Now the main threads signals the client threads to start measuring the execution times for 30 more seconds. When time elapsed the main thread signals the client threads to save their individual measurements inside a globally allocated standard vector. This procedure is guarded by a semaphore to synchronize concurrency. The last client thread signals the main thread to dump the vector into a text file which is the last step the client application takes. After that the evaluation has finished and its results are presented in Figure 4.1 and Figure 4.2. For the complete measurement of the performance of the address sanitizer memcached has been invoked with the environment setting `ASAN_OPTIONS=report_globals=0` to disable the reporting buffer overflow detection. This is necessary because this version of the thesis does not align global variables properly at certain boundaries and the address sanitizer reports this behavior as an error.

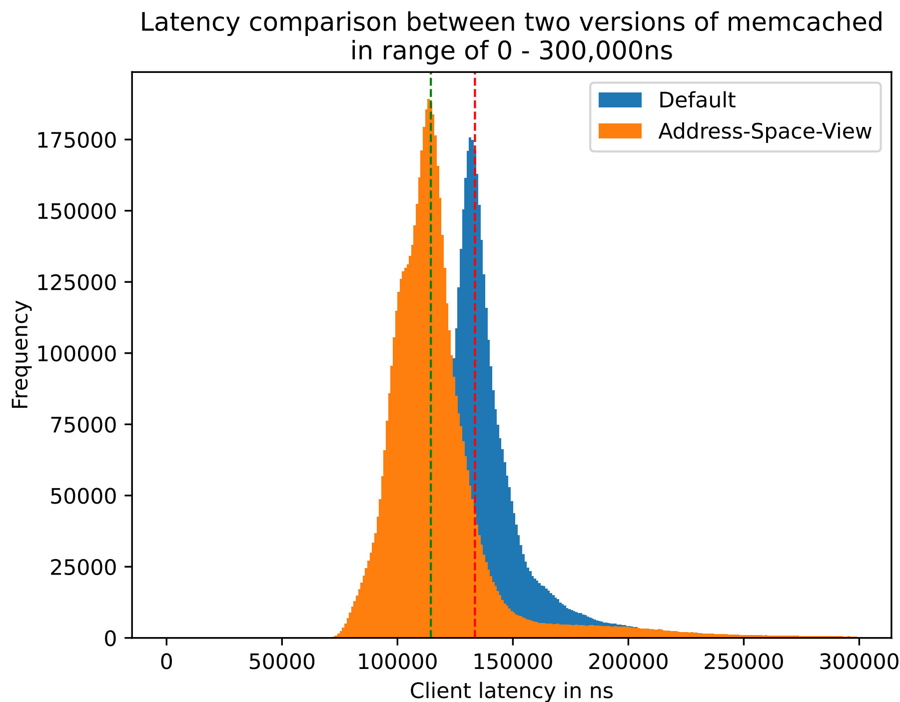


Figure 4.1 – Client latency histogram for the evaluated versions of memcached. The data displayed is in range of 0 to 300,000ns to highlight the highest peaks of both approaches. The green-dotted line marks the median of the ASV approach while the red-dotted line represents the median of the default version.

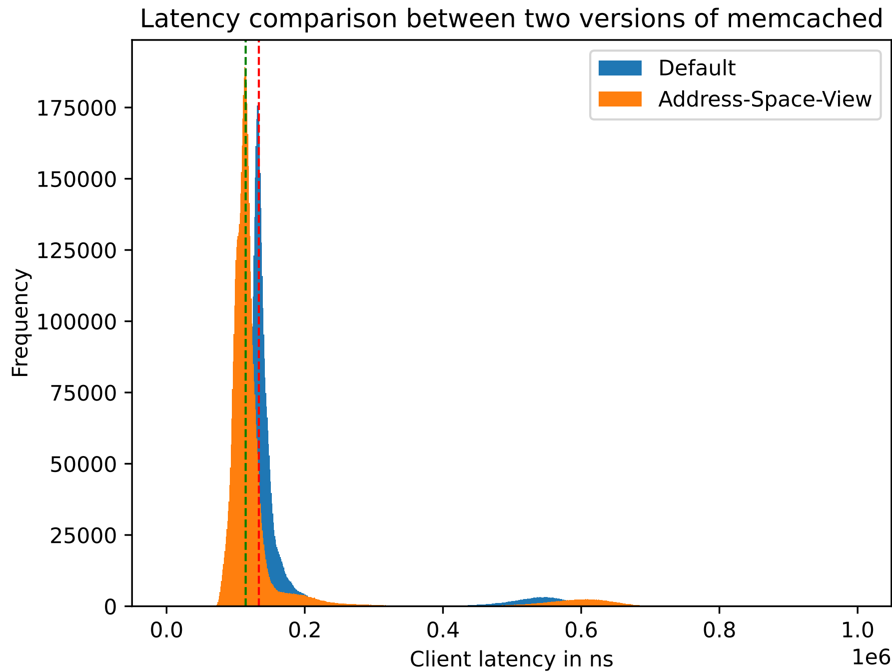


Figure 4.2 – Client latency histogram for the evaluated versions of memcached. The data set has been ordered ascendingly and the upper 1% has been cut off to avoid displaying potential outliers. The green-dotted line marks the median of the ASV approach while the red-dotted line represents the median of the default version.

Figure 4.1 shows the direct impact of the developed approach on the client latency. With the orange graph representing the histogram for the memcached version leveraging ASVs, the client latency can be reduced for the case of applying the address sanitizer to all functions of only one object file. The default version, which applies address sanitization to the whole application, reached higher client latencies in comparison to the thesis' approach. Using ASVs the client's median latency has been 114471ns while the default version reached 133560ns which means that the established approach reached a latency reduction of approximately 14.3%. Figure 4.2, however, shows that the default version of memcached reaches lower latencies around the range of 500,000 to 630,000ns compared to the Address-Space View variant. Potential reasons for that measure are highlighted and discussed in Section 5.4.

4.3 Summary

In this chapter the reader has been introduced into the evaluation of the thesis' approach and its impact on runtime execution. At first synthetic test cases showed the core functionality of the approach and prove that the ELF files can be extended to include several application views by creating dedicated sections. Furthermore, they also show that a runtime can be developed which is able to parse that information and apply the application view switch during execution in a specially developed test application.

For the second test case *memcached* as a real-world application has been chosen to evaluate the thesis' approach in an application which is widely used. The measurements do not indicate a significant positive impact of the taken approach to execution times of the micro-benchmarked function *process_get_command* for 25 keys. They do, however, indicate that the approach is robust and does not lead to application crashes for several million function executions. The third test case evaluates the systematic application of the *AddressSanitizer* to a specified object file in order to evaluate potential performance gains in cases where developers like to sanitize only certain parts of their application. The 32 worker threads measured the latency of continuous calls to the *memcached_get* library method which retrieves the same 25 keys as in the second test case. One version of memcached applies the sanitizer to its complete code base while the other version makes use of ASVs to only sanitize the object file in which that method is defined. The results provide a measurable effect of the established multivariant approach by reducing client latencies when utilizing ASVs for the majority of the clients calls by approximately 14.3%. They do also clarify that the default version of memcached can outperform Address-Space Views for higher latencies which is further discussed in Section 5.4.

5 DISCUSSION

The previous chapters explained the thesis' approach and evaluated its impact upon synthetic and real-world applications. Now the reader is introduced into a critical discussion about the implementation and its benefits and drawbacks. After this chapter he is able to draw a more reasonable conclusion on the influence of this thesis with respect to closing the gap between static and dynamic variability.

5.1 Increased code complexity in the linker

To enhance the lld linker to allow the building of multivariant ELFs its code base has been extended as described in Section 3.1. A primary goal of this project has been to work as minimal invasive as possible which is why all the added functionality resides in two extra files. During the linking process the lld imports the multivariant linking interface and executes its various functions on demand. All of these functions are wrapped in a dedicated `MVL` namespace to further increase distinguishability between the existing code base and the multivariant approach. At specific points in linking, such as symbol gathering, input section file parsing or section file offset generation, these functions are called in order to gather all the information needed to allow building multivariant executables.

A drawback of the established approach is that the code base is now bloated up for all the use cases in which the building of multivariant object files is undesired. Also, the lld itself is now built to always support multivariant linking which is not necessary for linking platforms in which only traditional linking occurs. Benefits of the developed approach are that even though the code complexity has increased its real complexity is hidden behind a namespace and the existing code has to call only a handful of new methods which arguably reduces the harm in readability. Because the consistent use of that namespace developers are able to identify the added function calls and can remove them from the code base if they are not needed. In fact the existing solution can be easily extended to make use of the C preprocessor to build the lld with multivariant ELF support only when it is desired.

All in all, even though the code complexity has increased the minimal invasive strategy remains the readability of the code and can be extended to build linker images which support multivariant linking if demanded.

5.2 Programming interface

Section 3.2 introduced the building of a runtime to apply memory overlaying to facilitate view switching during execution. Its programming interface contains the function `load_and_switch_view(id)` which takes an integer, loads an application view from the application image into a new Address-

Space View and switches to that view at the end. The implementation and method signature is straightforward, and the goal was to ensure the view switch can be technically applied. An index as a view selection does work, but requires the user of this runtime to exactly know which view lies at which index within the multivariant ELF. A more general approach can be to attach names to views as they allow to transport more information about that view to the user of the runtime. For example the existing approach in which a multivariant application has two views, index 0 and 1, can be changed to attach the names *default* and *no-log* to the views. With that change the readability of the code increases as the reader, especially a coworker or reviewer, can semantically follow the reasoning of the developer and figure out potential issues in comparison to sole integer indices.

5.3 Executable size overhead

Multivariant ELF files contain additional data which increases the size of the program image. This data includes the metadata and the additional sections/segments described in Section 3.1.4, as well as redundant data. Redundant are most of the data sections out of the multivariant object files, because all view instances only access one version of these sections at every point during execution, however all versions are saved within the final binary.

5.3.1 Meta data and view data

To facilitate multivariant executables this thesis relies on additional sections and segments to separate different views of the application. This enforces a larger Section Header Table and a larger Program Header Table where both of their entries require the amount of bytes described by the members *e_phentsize* and *e_shentsize* of the ELF header. The total increase in image size does also heavily depend on the amount of multivariant functions, as well as on the size of each of these functions and is therefore individual to every specific application setup which makes it hard to give an average estimate. Additionally, for the version of this thesis it is important *where* these multivariant functions are defined because the view granularity is at object-file level. The linker places all functions out of these object files into the final executable which means that even non-multivariant functions with the exact same code are saved twice if two multivariant views are generated. That means that applications that define two multivariant functions within the same object file are potentially smaller than applications that define these two across two different object files. In total the increase in size of the application image released by the view data can be calculated using the following formula

$$S_{view} = \sum_{m=1}^{j}(\sum_{n=0}^{k} size_{func}(m,n)) + size_{sht} + size_{pht} \tag{5.1}$$

where j describes the number of views and k denotes the number of multivariant functions. $size_{\text{func}}(m, n)$ is a function which returns the size of the *nth* function inside the *mth* view and $size_{sht}$, $size_{pht}$ return the sizes of an entry within the SHT or PHT, respectively.

The metadata includes all necessary information to find all multivariant view data and resides in its own section, too. It does not require an own segment but incorporates a dynamic symbol to point to its memory location. The content of the metadata is described in Section 3.1.4 and part of the following equation to calculate the its overhead size

$$S_{meta} = (\sum_{m=1}^{j} size_{int}) + 4 * size_{int} + size_{sht} + size_{sym} \tag{5.2}$$

where j is the number of file offsets of all multivariant output sections, each of them is saved as an integer. Additionally, $size_{int}$ is added four times because we require an integer for the size of a view, the number of views embedded in the ELF, the base virtual memory address for all views and the total size of the meta structure. After that the $size_{sht}$ is added because of its *.genmeta* section and $size_{sym}$ describes the size of the dynamic symbol which is an entry within the Dynamic Symbol Table and the String Table. The final image size increase can be calculated using $S_{total} = S_{view} + S_{meta}$.

5.3.2 Redundancy and view-local data

During the reading of Chapter 3 and for the reasons given above the reader may remember that the thesis' approach does not allow to define *view-local* data. To recapitulate view-local data refers to data that is not shared among other multivariant views of the application. This thesis requires all views to define and share all of their global data such as global variables, as well as statically defined data. It may, however, be desirable to define local data parts, as there are numerous examples and use-cases in which they are beneficial. One of the simplest applications of view-local data is when data is required by only one view and no other view. Other views then do not needed to know anything about this data as they do not operate on it anyways. Additionally, it hardens security if this data is hidden from other views as well. In fact *view-local* data can be compared to *thread-local* data concerning its benefits, drawbacks and use-cases. As thread-local data allows to define data which is private to each thread view-local data allows to define data which is private to each view so both of these concepts are strongly related to each other. Furthermore, view-local data eases synchronization issues across multiple views as the current state of this project synchronizes all multivariant data and the programmer has to be aware of that concurrency. Synchronization issues have been prominent in concurrent applications for decades and with the current approach another layer of that issue is created.

The term *redundant* is used because every view comes with its own data but as Section 3.1.3 and Section 5.3 describe all relocations lead to data accesses to only one version of that data across all views. That means that all the other data sections of the views are not accessed during execution and are therefore redundant. Because view-local data is not supported by the thesis that data serves no purpose and can be omitted and is part of possible future work as Section 5.5 will explain. Other redundant data are the functions inside the multivariant object files which are not multivariant on their own. Since these functions are the exact same in all views they can be omitted as long as one of their versions is kept inside the executable but outside a memory overlay unit so that it is present in each and every view all the time.

5.4 Performance impact

The multivariant implementation does not only induce space but also runtime overhead as the view exchange mechanism increases latency by additional program logic and its side-effects such as cache pollution. It is now of great interest to see whether the potential performance benefits of dedicated application views can overcome their cost to make an educated decision in which scenarios the thesis' approach is a benefit to the application and in which it is not. Keep in mind, however, that there are more potential benefits than only performance gains with *security* being one of the discussed ones in Section 5.5.

Dedicated application views being wrapped in different ASVs provide potential performance benefits by reducing the amount of executed instructions if the application decides to switch to a view which contains less logic by omitted logging calls for example. Omitting these calls can also

be achieved by wrapping them around conditional logic which is evaluated during runtime. That means that for two scenarios, one using multivariant views and the other using conditionals, what really can be saved are the execution and evaluation of these conditionals e.g. if-statements. Today's computing systems contain many concepts to forecast these conditional branches, with the help of a branch predictor for example. In combination with speculative execution the CPU executes that code that it expects to run when a certain condition is met and either applies their results or discards them and continues with the other part of the branch. If the forecasting system has a high hit rate then the saving of omitting the evaluation of conditionals will not lead to significant latency decreases. Caching mechanisms reduce these performance benefits as well which explains the small to no performance gain of the micro-benchmark in Section 4.2. Additionally, the application of Address-Space Views requires to flush the Translation Lookaside Buffer (TLB), an important memory mapping cache, because the switch between ASVs is effectively a context switch [15, p. 6]. Flushing the TLB further mitigates potential performance benefits.

Another latency impact is the dynamically linked runtime library as it requires a trampoline over the Procedure Linkage Table (PLT) to call its functions. Especially for scenarios in which a thread decides to switch into a view in which it already is, these calls are expensive as the called function will not execute any further logic besides to not switch and return back to the caller.

What can increase the performance, however, is the application of dedicated views when it comes to sanity-checking. As Figure 4.1 clearly shows there can be a performance benefit to client latency when the sanity-check is only applied where it is needed. This circumstance is useful if a target application faces recurring problems and further investigation is needed. The developer can start the application and directly switch to the sanity-view or have that view be the default one in order to sanitize only the necessary parts of the application. With that it is possible to get verbose information and to target that to one thread rather than having the whole application and all of its threads to be in that verbose application view state all the time. As shown in Section 4.2, the common client latency can be significantly reduced by around 14% when using a dedicated sanitization view in memcached.

Figure 4.2 teaches about potential drawbacks of the thesis' implementation for higher client latencies. The mindful reader will notice that these latencies are up to five times higher than most of the measured latencies which is a remarkable deviation. Context-switches have hugh impacts on all types of application latencies as they defer the processing of the current thread to a later point in time. While this ensures that every thread of the system can progress, it is still limiting latency of each thread on its own. Address-Space Views suffer stronger from context-switches as a switch from one application thread to another application thread induces a TLB flush if these two threads reside in different views. Usually all application threads share the same address space and therefore a context-switch between these types of application threads is cheaper compared to ASVs. The two peaks around 500,000ns to 650,000ns are most likely measurements including these context-switches which can explain why the developed approach reaches higher tail latencies than the default one. To prove that context-switches are the reason behind increased client latencies and why the default version of memcached has been faster than the ASV version for high latencies both of these versions should have their threads pinned to specific CPUs. The pinning of certain application threads to certain CPUs reduces context-switching overhead. When the threads are pinned it is less likely that one thread of the same application will be replaced by another one with a different ASV. The evaluator has to ensure that application threads of the multivariant executable which reside in different ASVs should not be scheduled on the same CPU. Although that is not completely configurable it is possible to set thread specific affinity masks. By using affinity masks the developer pins all threads according to their views. If the hypothesis proves to be true the higher

latencies of the ASV version of memcached should converge to the higher latencies of the default version because the TLB is not flushed as often.

After all it is important to notice that Figure 4.1 covers the majority of the data set for both histograms. The data displayed represents up to 92.72% of the default and 93.94% of the ASV total data set and exemplifies that even though there can be a negative performance impact of the thesis' method most of the time it's beneficial to client latencies for the case of address sanitization.

5.5 Future work

The core purpose of this thesis was to develop a platform to build multivariant applications by using dedicated Address-Space Views. Because this is a novel approach of facilitating multivariance it provides multiple aspects to improve on.

First of all the granularity of multivariant functions can be decreased from object file level to function level. As Section 5.3.2 explained multivariant views contain functions which are all the same in each and every view, thus increasing binary file size, process memory size and have a detrimental impact on caching systems as equal data is reloaded continuously. Therefore, it can be of great interest to filter out only those functions which are really desired to be multivariant. The developer should be able to *mark* multivariant functions in order to tell the linker which functions are part of a multivariant view and which are not. The existing view building mechanism can be easily extended to not put all functions of a multivariant object file into an output section *.gen* but rather skipping those being not marked as such.

Speaking of marking another important aspect to improve is the implementation of *view-local* data. Currently, no view-local data can exist which first can be a feature developers would like to use like they may use TLS and second results in redundant data being placed in the final executable. The drawbacks of object level granularity apply to view-local data too and provide a great surface for improvement. Again the developer needs to be able to mark view-local data objects such as global variables so that the linker can manipulate relocations accordingly to allow different views to access different data which is then not synchronized between views. Both of these improvements contribute to harden *security* in multivariant applications. The fact that a view can be tailored to especially one thread allows to include several hierarchies of responsibilities and permissions. The developer can define dedicated threads being responsible and being the only ones allowed to execute certain tasks such as making database calls or parsing user information. If the view is highly restricted that secures the whole system's environment by mapping only the code that is necessary to perform the work the thread has to do. The idea of building restricted, isolated views to perform certain work, however, is not new and existing research already elaborated on it. James Litton et al. describe *light-weight contexts (lwCs)* as isolated address spaces to which each thread can individually switch to in order to execute program logic in the fashion of coroutines[9, p. 5]. This approach of organized, controlled work scheduling is similar to the idea of restricted views with view-local data, but the semantics of both of these quite differ. While this thesis focuses on a strong synchronization between the ASVs with dedicated diverging areas, lwCs targets to provide contexts in a fork-like manner by even creating copies of file descriptor bindings[9, p. 1]. Also the concept of coroutines being applied to the latter is not of particular interest to the former. Ultimately, with the approach established over the course of this thesis, the depth of security is a derivation of the developer's creativity and expertise of conceiving restricted views.

Another small improvement is to enable static linking against the developed runtime library. Using a static link the linker can inline the function calls to the library, thus saving the call and the indirection over the PLT jump table to further reduce latency. In order to inline functions inside

the linker it has to support Link Time Optimization (LTO) and the llvm lld used by this thesis does support it [11].

Last but not least, the multivariant linker and runtime frontends can be optimized to improve their usability. The linker includes the *--mvo and --multivariant-linking* argument switches to control the linking process and with the approach given above, by explicitly marking functions, the linker can almost omit these arguments. If it is feasible to mark certain function variants as default variants the arguments can be left out completely, otherwise the *--mvo* switch is needed to configure the default view. For the runtime, a more human-readable and reasonable form of switching the view can be established by moving from unclear integer indexes to named views as proposed in Section 5.2.

5.6 Summary

In this chapter the reader followed a discussion about the results and evaluation of this thesis in order to be able to critically think about the taken approach and its benefits and drawbacks. She understands that the implemented code in the lld linker increases its complexity, especially for the case in which the developer does not want to make use of multivariant linking. The code base, however, contains a dedicated *MVL::* namespace to reduce the induced complexity and to allow developers to distinguish between the existing code and the added one. Next to the existing code base a new runtime library has been developed which provides a minimal interface to switch between different views. Instead of passing integers, named views increase the comprehensibility of the runtime and ease its usage.

Beside the increasing code bases the final binary ELF increases in size as well due to the additional sections and segments. The reader learned that the meta information as well as the redundant view data imply that drawback even though the metadata is necessary to allow the runtime to find its information. What can be omitted, however, is the duplicated unused data since all views access the same data. That applies to duplicated non-multivariant functions in all views as well and Section 5.5 suggests enabling the marking of functions and certain data to reduce redundancy and to introduce *view-local* data.

Finally the evaluation has shown no significant performance impact on a micro-benchmarked function. For the macro-benchmark of client latencies, however, it showed that dedicated views with only the necessary address sanitizing enabled can have a positive impact on the execution time and provides the reader with two scenarios in which both, the developed approach can and cannot be beneficial to the underlying application. The reason why the established method can also be detrimental to client latencies has to be further evaluated. This thesis suggests to investigate the impact of context-switches upon client latencies to explain that increase. For the case of address sanitization in memcached this thesis was able to close the gap between static and dynamic variability by transporting the performance benefits of a statically tailored sanitized version of memcached to its runtime environment.

6 CONCLUSION

Over the course of this thesis a new approach of multivariant applications has been developed and evaluated. That approach embeds dedicated application views inside the ELF executable and switches between those during runtime on demand. It achieves that by extending the llvm linker to build sections and segments of multivariant functions. Each of these segments are of the same size and contain the functions built for their views. The views are memory overlayed and can be exchanged during runtime with the help of the developed runtime overlay manager which creates *Address-Space Views* for every embedded view. The application of the view switching is thread-local so that one switched thread executes one view variant while all other threads remain in their views. Multiple threads can reside in the same view, thus sharing their instruction code and data objects.

The evaluation shows that it is possible to create and switch to multiple application views in synthetic test cases as well as in the real-world application *memcached*. Furthermore, they reveal that the new approach can lead to significant performance improvements in certain application use-cases such as address sanitizers. For memcached a median client latency reduction by 14.3% can be achieved for the optimized use of the address sanitizer. But the test cases also show that this impact can be neglectable or detrimental for scenarios in which the cost of Address-Space Views overshade omitted instructions. The reason why ASVs can have a negative impact on client latency has to be further evaluated, but a particular point of interest can be to investigate the increased cost of context-switching for these views. This circumstance exemplifies that the thesis' approach does not provide benefits to performance in all manners. It is strongly tied to the application use-case and can significantly improve latency for certain scenarios.

Beside performance improvements *multivariant ELFs* also contribute to hardened security in multi-threaded applications. By providing dedicated tailored views, the developer is able to restrict a thread's code to an absolute minimum. That restriction enables a new layer of protection and can even build a hierarchy of authorization in which certain threads are allowed to execute privileged code while others cannot because its not mapped into their view. Further work on this project can introduce the concept of view-local data to enable a more fine-grained data access in views. View-local data is not synchronized across view instances and provides developers with more control over data accessibility inside the views. Additionally, it further hardens security as certain views can access confidential data while that data is hidden from all other views at the same time. Finally, an optimized runtime overlay manager can reduce latency by skipping the indirection over the PLT with static linking instead of dynamic linking and by making use of function inlining.

In conclusion, multivariant ELF executables and Address-Space Views together provide the possibility to leverage thread-specific multivariance computer applications by utilizing tailored application views to increase security or to gain performance benefits as this thesis has shown for use of address sanitization.

LIST OF ACRONYMS

ASV	Address-Space View
COW	Copy-on-write
CPP	C preprocessor
CPU	Central Processing Unit
ELF	Executable and Linkable Format
LTO	Link Time Optimization
lwC	light-weight context
PHT	Program Header Table
PLT	Procedure Linkage Table
RAM	Random Access Memory
SHT	Section Header Table
TLB	Translation Lookaside Buffer
TLS	Thread Local Storage
VA	Virtual Address

LIST OF FIGURES

2.1 Chain of thought for related data, sections and segments. Data that belongs together is aggregated into sections. Sections are then aggregated into segments. Segments do typically not have a name associated. Clouds represent types of data, circles represent sections and squares represent segments. 5

2.2 An overview of an ELF and its content. It starts with a header further describing its different types of data such as the PHT and the SHT. The PHT holds information about all *segments* in the ELF. Segments do not have a name associated, but can be distinguished by their access rights: $r = read, w = write, x = execute$. Within a segment different *sections* are placed. These sections aggregate information of equal semantic meaning such as code or data for example. 5

2.3 Basic visualization of generating an executable file out of multiple input source code files. Every file is translated into an *object file* by a *compiler* and then linked together by a *linker* to produce the output executable. 6

2.4 Visualization of a symbol table lookup. The *linker* can parse the ELF Header to find the SHT. Every entry within the SHT holds a type which identifies its content. The magic value of *SHT_SYMTAB* tells the linker that this section holds the symbol table. The attribute *sh_offset* holds the file offset within the binary where the symbol table is stored. 8

2.5 Symbols and relocations. Together they allow dependency resolution by the linker. The relocations in *main.c* reference an undefined symbol *sum*. This symbol is defined in *foo.c* and can therefore be resolved during linking where the *linker* assigns addresses to *data objects*. It assigns the address *0xc0ffee* to the data object *sum* and can now relocate the calls of *main.c* at file offset 20 and 178. After relocation the calls to *sum* at addresses 0x60000 and 0x70000 succeed. 9

2.6 Memory overlaying concept. The area to overlap here is described as the sections *.mem-overlay1* and *.mem-overlay2*. During the runtime of the application *.mem-overlay1* is loaded. At some point during execution *.mem-overlay2* is laid over *.mem-overlay1* thus replacing it. References of *.mem-overlay1* and *.mem-overlay2* into *.text* must remain valid at each moment of execution. 10

2.7 Two ASVs next to each other. Most of the data between these two views is shared, but certain parts can be *pinned* which results in an own private COW memory mapping. The pinned part is marked in green. Figure taken out of [15, p. 5]. 12

3.1 Symbol resolve for multivariant object files. The function symbol *db_get_user* is defined in *foo1.o* and *foo2.o* and declared in *main.o*. *db_get_user* is bigger in *foo2.o* because it contains logging logic which *foo1.o* does not. The linker argument call lists the statement *--mvo 0:foo2.o --mvo 1:foo1.o*. When the linker resolves that dependency in a multivariant scenario it takes the definition that is given as index 0 in the linker argument list for the *--mvo* linker argument. The result is all declarations of *db_get_user* are connected to the definition within *foo2.o*. The definition within *foo1.o* is included in the executable, but never called. 18
3.2 Comparison of two function variants of *db_get_user* and *db_get_order*. Code is drawn as blue-shaded area, padding is grey-shaded. These functions are part of a memory-overlayed unit, so they must have the same size. The smaller function definitions contain padding bytes to reach the same size as their bigger counterparts in the other object file. 19
3.3 Multivariant segments and output sections in the ELF file. The main executable has two additional read/execute segments colored in purple. The first purple segment behaves as the default segment, it is loaded initially during application startup. Both segments contain an output section *.gen* which bundles all input function sections of a multivariant object file. During runtime the first segment can be overlaid by the second segment to switch from one application view to another. 20
3.4 Relocations of local symbols of two multivariant object files *foo1.o* and *foo2.o*. Each object file contains a *.rela.text* section which contains all relocations for the *.text* section. The references of these relocations are towards the local symbols of their own *.rodata* section prior linking. During linking the linker redirects these references to the symbol definitions of *foo1.o* because its view index has been 0. In the end that leads to *.text - foo2.o* accessing not its own data but the data of *foo1.o* so that both views refer to the same data objects. 21
3.5 Final layout of a multivariant ELF. *foo1.o* and *foo2.o* are multivariant object files which both contain a function definition for *db_get_user*. Before linking the relocations for *db_get_user* are directing towards the object file's individual read-only data sections. For the final executable the linker created two new sections, *.gen1* and *.gen2*, which both express two views within the application. Each of them contains a variant of the function and the linker made sure that both views access the same data objects by manipulating the relocation entries of *foo2.o*. The *.genmeta* section contains the meta information necessary to identify the multivariant sections during runtime. The sections of *main.o* have been omitted due to lack of space. 23

4.1 Client latency histogram for the evaluated versions of memcached. The data displayed is in range of 0 to 300,000ns to highlight the highest peaks of both approaches. The green-dotted line marks the median of the ASV approach while the red-dotted line represents the median of the default version. 31
4.2 Client latency histogram for the evaluated versions of memcached. The data set has been ordered ascendingly and the upper 1% has been cut off to avoid displaying potential outliers. The green-dotted line marks the median of the ASV approach while the red-dotted line represents the median of the default version. 32

6.1 Visual representation of Table 6.1 and Table 6.2. 54

LIST OF TABLES

6.1 Execution time per 1.000.0000 function calls over five iterations and four threads, in nanoseconds. The function does print the keys. 53
6.2 Execution time per 1.000.0000 function calls over five iterations and four threads, in nanoseconds. The function does not print the keys. 53

LIST OF LISTINGS

3.1 Usage of the C preprocessor. Program code between *#ifdef...#endif* regions are translated if the given name is defined. In this example the printf function call will be translated because LOG is defined in line 1. 16
3.2 Example code for linking with the new linker options. 16

4.1 Synthetic test case example code. 27
4.2 Command to link the multivariant executable. The listing of important standard libraries to link against are omitted, the focus is on the new aspects of the linker interface. 28
4.3 Synthetic test case example code. 28
4.4 Command-line output of the view switching test case. 29

REFERENCES

[1] Steve Chamberlain. *Using LD, the GNU linker - Overlays*. https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_22.html, accessed: 11.08.2021.

[2] *Computer Science from the Bottom Up - Chapter 9. Dynamic Linking*. http://bottomupcs.sourceforge.net/csbu/x3735.htm#AEN3751, accessed: 09.08.2021.

[3] Robert C. Daley and Jack B. Dennis. "Virtual Memory, Processes, and Sharing in Multics." In: *Proceedings of the First ACM Symposium on Operating System Principles*. SOSP '67. New York, NY, USA: Association for Computing Machinery, 1967, 12.1–12.8. ISBN: 9781450373708. DOI: 10.1145/800001.811668. URL: https://doi.org/10.1145/800001.811668.

[4] GCC docs. *The C Preprocessor: Conditionals*. http://gcc.gnu.org/onlinedocs/gcc-3.0/cpp_4.html, 4.2 Conditional Syntax, accessed: 20.08.2021.

[5] Dormando. *memcached - a distributed memory object caching system*. https://memcached.org/, accessed at 03.09.2021.

[6] Elinux.org. *Executable and Linkable Format(ELF)*. https://elinux.org/Executable_and_Linkable_Format_(ELF), accessed: 03.08.2021.

[7] Bryan Henderson. *Linux Loadable Kernel Module HOWTO*. https://tldp.org/HOWTO/Module-HOWTO/x73.html, 2. Introduction to Linux Loadable Kernel Modules, accessed 04.10.2021.

[8] ORACLE Linker and Libraries Guide. *Symbol Table Section*. https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-79797.html, accessed: 09.08.2021.

[9] James Litton et al. "Light-Weight Contexts: An OS Abstraction for Safety and Performance." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, 49–64. ISBN: 9781931971331.

[10] LLVM. *AddressSanitizer - Clang 13 documentation*. https://clang.llvm.org/docs/AddressSanitizer.html, accessed: 06.10.2021.

[11] LLVM. *LLVM Link Time Optimization: Design and Implementation*. https://llvm.org/docs/LinkTimeOptimization.html, accessed: 17.09.2021.

[12] Linux manual page. *elf - format of Executable and Linking Format (ELF) files*. https://man7.org/linux/man-pages/man5/elf.5.html, accessed: 09.08.2021.

[13] Linux manual page. *ld - The GNU linker*. https://man7.org/linux/man-pages/man1/ld.1.html, accessed: 09.08.2021.

[14] Linux manual page. *mmap(2) - Linux manual page*. https://man7.org/linux/man-pages/man2/mmap.2.html, DESCRIPTION, accessed: 29.08.2021.

[15] Florian Rommel et al. "From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes." In: *14th USENIX Symposium on Operating Systems Design and Implementation* (2020), pp. 36–72.

[16] Florian Rommel et al. "From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes." In: *14th Symposium on Operating System Design and Implementation (OSDI '20)*. 2020, pp. 651–666. URL: https://www.usenix.org/conference/osdi20/presentation/rommel.

[17] Florian Rommel et al. "Multiverse: Compiler-Assisted Management of Dynamic Variability in Low-Level System Software." In: *Fourteenth EuroSys Conference 2019 (EuroSys '19)* (Dresden, Germany). New York, NY, USA: ACM Press, 2019. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303959.

[18] Tool Interface Standards(TIS). *Executable and Linkable Format (ELF)*. http://www.skyfree.org/linux/references/ELF_Format.pdf, accessed: 03.08.2021.

[19] Wikipedia. *Magic number (programming) - Format indicators*. https://en.wikipedia.org/wiki/Magic_number_(programming)#Format_indicators, accessed: 03.08.2021.

APPENDIX

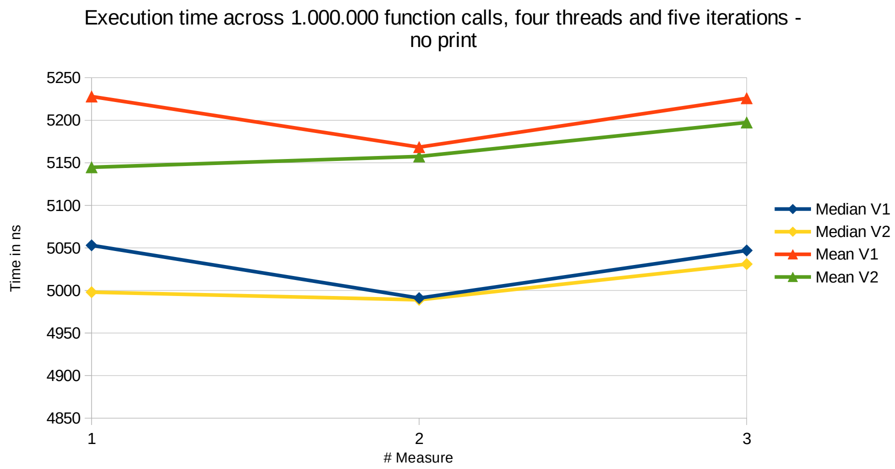
The following tables represents micro-benchmark measurements for get requests of 25 keys to a memcached server. Version 1 refers to the version of memcached using if-statements rather than address space views. Version 2 makes use of address space views and omits if-statements to toggle printing. Table 6.1 includes execution times for a scenario in which *process_get_command* does output the keys whereas Table 6.2 measured the same method without the keys being printed.

	# Measure	Total Median	Total Mean
Version 1	1	170221	194530.67
	2	169716	193481.18
	3	169652	194065.01
Version 2	1	173327	200241.42
	2	172009	197077.08
	3	175916	204190.98

Table 6.1 – Execution time per 1.000.0000 function calls over five iterations and four threads, in nanoseconds. The function does print the keys.

	# Measure	Total Median	Total Mean
Version 1	1	5053	5227.90
	2	4991	5168.41
	3	5047	5225.80
Version 2	1	4998	5144.74
	2	4989	5157.46
	3	5031	5197.39

Table 6.2 – Execution time per 1.000.0000 function calls over five iterations and four threads, in nanoseconds. The function does not print the keys.



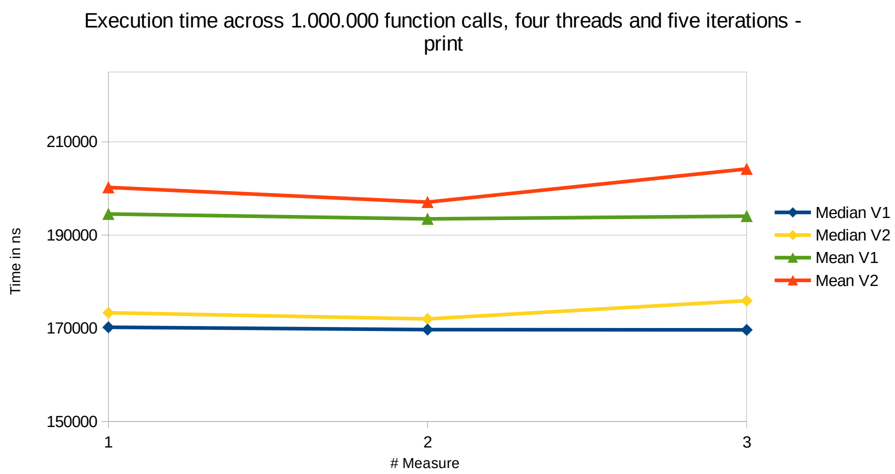


Figure 6.1 – Visual representation of Table 6.1 and Table 6.2.