MASTER'S THESIS | LUND UNIVERSITY 2016

LLVM-Based Fortification for Kernel Drivers

Caroline Brandberg

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-28

LLVM-Based Fortification for Kernel Drivers

Caroline Brandberg

August 8, 2016

Master's thesis work carried out at SYSGO AG, Germany.

Supervisor: Dr. Jonas Skeppstedt
Dr.-Ing. Henrik Theiling

Examiner: Asst. Prof. Flavius Gruian

Abstract

In today's operating systems, drivers are linked with the kernel where handling pointers and performing memory accesses must be considered with much more care than in application user space.

This thesis focuses on two issues. First, memory access to user space must never be done directly, because the access may fault due to insufficient access permissions or unmapped pages. Second, pointers entering via system calls must be checked prior to their use to prevent a malevolent user from exploiting kernel drivers to access kernel space for them.

The proposed solution uses the type system of Clang combined with analyzes on the generated LLVM intermediate representation, both in the purpose of performing static analyzes to produce valuable messages to developers during compile time, but also to insert robustness assertions and perform code transformations. With these precautions we were able to identify four bugs in a single device driver.

Keywords: LLVM, Device Drivers, Kernel Memory Access, Pointer bugs

Acknowledgements

First, I would like to thank Henrik Theiling for giving me the opportunity to write my Master's Thesis in such an inspiring topic. I'm grateful for his sincere support, knowledge, enthusiasm and professionalism that he constantly provided me with during my time.

I would also like to thank SYSGO for the opportunity to work in an inspiring workplace with such high competent employees. I especially want to thank Andrea Bastoni, Marco Kosinski, Andreas Schik, Konstantin Tarandevich and Andreas Betz.

My main supervisor, Jonas Skeppstedt, for being an inspiring person, both during my Master education as my teacher and during my thesis as my main supervisor. Also, thank you Flavius Gruian for being my examiner.

Finally, I would like to thank my family, Jan Brandberg, Marita Brandberg, Fredrik Brandberg and Johan Dahlgren for always being my biggest support and for proof-reading my work and providing me with valuable hints.

Contents

1 Introduction **11**
1.1 Related work 12
1.1.1 Pointer bugs in Kernel Drivers 12
1.1.2 LLVM 13
1.2 Structure 13

2 Background **15**
2.1 Types in C language 15
2.1.1 Pointers 15
2.1.2 Structures 16
2.1.3 Unions 17
2.1.4 Type Conversions 17
2.2 Operating System concepts 17
2.2.1 Terminology 18
2.2.2 Address spaces 18
2.2.3 Device Drivers 19
2.2.4 System calls 20
2.2.5 Security issues 21
2.3 LLVM 25
2.3.1 LLVM Overview 26
2.3.2 LLVM Intermediate Representation 27

3 Train of Thoughts **31**

4 Dynamic runtime checks **35**
4.1 Direct memory access to user space 35
4.2 Implementations 36
4.2.1 Call external function 36
4.2.2 Inline instructions 39
4.3 Measurements 41

4.4 Result 43

5 Static analyzes 47

5.1 Accessing kernel space on behalf of user space 47
5.1.1 Introducing new pointer types 48
5.1.2 Introducing new functions 50
5.1.3 LLVM Pass 52
5.2 Implementations 56
5.2.1 Introducing the LLVM-based fortification into PikeOS 56
5.2.2 LLVM Passes 56
5.2.3 Using the extension 58
5.3 Code examples 59
5.3.1 Correct usage 59
5.3.2 Limitation of providing the wrong aggregated variable 61
5.3.3 Limitations of cast operations 62
5.3.4 Limitation of unions 63
5.3.5 Limitation of function pointer 64

6 Discussions 67

6.1 Dynamic runtime checks 67
6.2 Static analyzes 68

7 Conclusions and Future work 71

7.1 Conclusions 71
7.2 Future work 72

Appendix A Data from measurements 79

Appendix B CODEO extensions 83

B.1 Call external function 83
B.2 Inline instructions 84
B.3 Changing type through dedicated functions 85
B.4 Members are marked unchecked 86
B.4.1 Insert checks automatically 88
B.5 Direct memory access 89
B.6 Integer to pointer 90

List of Figures

2.1 Pointer illustration where a pointer z is pointing variable x 16
2.2 User- and kernel virtual address space 19
2.3 PikeOS kernel driver API 19
2.4 Overview of how user space can enter data as parameters to system calls, which can later be accessed by the kernel 21
2.5 Simplified call graph for the PikeOS function drv_memcpy_in 24
2.6 Copy data from user- to kernel space 25
2.7 The LLVM compiler steps 26

3.1 This works main concerns with a flow chart of the founded threats and the proposed countermeasures 32

4.1 Inserting LLVM instructions which will generate a call to an external function, which in return will ensure that there are no direct accesses to user space. 38
4.2 Creating Basic Block A and B 39
4.3 The final created block situation 40
4.4 Difference between various techniques used for the external method . . . 43
4.5 Performance increase of the two different techniques: inlining instructions, and performing the check in an external function on x86 44
4.6 The impact of excluding checks of local- and global variables 44
4.7 Performance increase of the two different techniques: inlining instructions and performing the check in an external function on ARM 45

5.1 Division of different pointer types, illustrated in a tree structure 48
5.2 Pointer flow in PikeOS kernel drivers 50
5.3 Data provided via a system call 52
5.4 Data provided from another device driver 52

B.1 Enable LLVM pass which fully prevents direct accesses to user space by invoking an external function 84

B.2 Example code which performs a direct access to user space 84
B.3 Generated runtime error that prevent direct access to user space, that is prevented by invoking an external function 84
B.4 Enable LLVM pass which fully prevents direct accesses to user space by inlining LLVM instructions 85
B.5 Generated runtime error that prevent direct access to user space, that is prevented by inlining LLVM instructions 85
B.6 Enable LLVM pass which checks that changes between the introduced types are done through the introduced functions 86
B.7 Example code which changes the introduced types by performing simple cast operations, i.e. not through the introduced functions 86
B.8 Enable LLVM pass which checks that all members are marked unchecked when data is copied in 87
B.9 Example code which copies in data that contains members that are not marked unchecked 87
B.10 Enable LLVM pass which automatically perform proper checks on all members that are marked unchecked when data is copied in 88
B.11 Example code which access members which have not yet been checked . . 89
B.12 Generated runtime error that prevent accessing data which is incorrect . . 89
B.13 Enable LLVM pass which checks that direct memory accesses are only performed with pointers marked kernel 90
B.14 Example code which performs a direct access to user space 90
B.15 Enable LLVM pass which checks that cast from integer to pointer is only performed if the pointer are marked unchecked 91
B.16 Example code which performs cast operation from an integer to a pointer, where the pointer is not marked unchecked 91

List of Tables

2.1 Generated LLVM IR information from `struct` and `union` types 30

4.1 Overview of the different features for the conducted measurements on the external function 42
4.2 Overview of the two dynamic techniques used to prevent direct accesses to user space 42
4.3 Increase in time, measured as percentage of a single queueing port read- and write operation with various message sizes (2 – 2048 bytes) on x86 . 45
4.4 Increase in time, measured as percentage of a single queueing port read- and write operation with various message sizes (2 – 2048 bytes) on ARM 46
4.5 Size of the executables, number of inserted instructions and global variables in LLVM IR 46

5.1 Overview of the operations which can be performed on the three different variables created in the code example presented in this section. The table illustrates which operations are valid, which are prevented by the C type system, which are prevented with created analyzes and where there still are limitations 63

A.1 Measured time, in nanoseconds [ns], of a single queuing port read- and write operation on x86 with different message sizes, in bytes. 80
A.2 Measured time, in nanoseconds [ns], of a single queuing port read- and write operation on ARM with different message sizes, in bytes. 81

Chapter 1
Introduction

Research at Stanford University presents that 50-65% of the security defects in the Linux kernel are in the drivers [3]. Since approximately 70% of the kernel is drivers, it's reasonable that most bugs appear in them. But, even when looking at the relative error, i.e. dividing the error rate in the drivers with the error rate in the rest of the kernel, drivers have up to 7 times higher error rate [4] which shows that drivers need extra concern.

These concerns are of special importance when dealing with domains which require a high level of safety and security, such as aerospace, defense and automotive. Amongst others, these domains are the marketing target of the microkernel-based Real Time Operating System PikeOS [24] [29], which is used as a proof of concept in this work.

There have been several attempts to secure the kernel from vulnerable drivers where one of the earliest is the micro-kernel design, where drivers are extracted from kernel to user space. However, the increase in the security came with a decrease in the performance [10, 12]. Because of that, many drivers were kept in kernel space which is still the case today.

In case of implementing a driver in the kernel space, one must take certain precautions. Especially since this implementation means that additional code is introduced into supervisor mode, where handling pointers and performing memory accesses must be considered with much more care than in application user space. The kernel has no easy means to protect kernel driver developers against common bugs, and some bugs are unexpected to unexperienced developers, because the programming environment can be quite different. In particular, pointers passed in from user space via system calls need very special care.

There are the following concerns:

- Memory access to user space must never be done directly, because the access may fault due to unmapped pages or insufficient access permissions.
- Memory addresses must be checked for actually pointing to user space to prevent malevolent users from exploiting kernel drivers to access kernel space for them.

An unexpected memory fault in the kernel will in most operating systems cause the system to panic, because there is no way to know how to handle such situation safely. Use of dedicated access functions is thus mandatory. The problem is even more important when dealing with fully preemptive operating system also in the kernel, where no check prior to a memory access can ever guarantee that the actual access will be fault-free, because memory mappings may change in between the check and the actual access.

Especially for the first concern, drivers will seemingly work correctly even without the special precautions, but silently have very serious bugs that the kernel is unable to detect. A requirement-based functional testing may not detect the bugs we are concerned with, because to trigger a problem, applications would have to explicitly pass illegal pointers into the kernel. This may lead to low-quality code to be assumed correct. Due to the special programming environment in the kernel and the very easy ways to do memory accesses in C, even in code reviews, such lapses of care might not be spotted. Also, since unmapping and remapping is time critical, if the poor code crashes due to map accesses, it is highly likely to be in corner case situations that are very difficult to debug.

The proposed solution uses the type system of Clang combined with analyzes and code transformations on the generated LLVM intermediate representation to prevent the presented concerns. The prevention will be both in form of inserting robustness assertions as well as performing static analyzes to generate valuable information during compile time.

The main questions focusing on in this work are the following:

- How will the device drivers be affected by these precautions, especially in terms of performance overhead?
- Is it possible to provide a watertight solution?

1.1 Related work

1.1.1 Pointer bugs in Kernel Drivers

Several attempts have been conducted to prevent bugs related to user/kernel space pointers. Some of these approaches propose to identify them through type qualifiers [9, 14, 32, 31]. Johnson and Wagner [14] present a technique using CQUAL [11] which found 17 user/kernel pointer bugs in the Linux kernel. This technique was later applied to the FreeBSD 5.3 kernel, identifying 5 user/kernel pointer bugs [9]. Another approach is the famous Sparse which is a semantic parser for the C language initially stated by Torvalds [31]. Sparse distinguishes user and kernel pointers by attaching an attribute, relating them to different address spaces, which is an inspiration for the static analyzess part of this work.

Other techniques have also been applied. Bugrara and Aiken present a static analyzing technique to find bugs also related to the problem, namely unchecked user pointer dereferences [2]. S. Peiró et. al. [23] also use a static analyze technique to detect information leaks of the kernel. Further, the tool Coccinelle [20], which is a control-flow based tool, which also has been applied to the Linux kernel to prevent these kind of bugs.

All of the known approaches have been contributing to making the drivers more secure. Although, to our knowledge, these approaches either separate kernel/user pointers or unchecked/checked user pointers. Both of these separations are of huge importance and therefore, a combination would be ideal.

1.1.2 LLVM

The LLVM [18] project has been used for a wide area of different code transformations, analyzes, etc., also for security [5, 8, 7, 33].

D. Dhurjati et al. [8, 7] uses LLVM to provide memory security without runtime checks, annotations or garbage collection. Although, the novel approaches presented in both papers are only applicable for programs using a restricted subset of the C language.

Another work is Emscripten [33], which translates LLVM Intermediate representation into JavaScript. The aim of the work is to enable code for the web initially written in various languages. The translation is especially to asm.js [13], which is a subset of JavaScript. The browser which will then run this code also needs to fortify it and virtualize all memory accesses in software, so that distinct browser windows will be unable to overwrite each other or the browser code or data. This shows that running fortified native code is feasible also in the presence of security concerns.

This work applies the LLVM technique to memory protection in the special situation in supervisor code, i.e. where there are no security net as for user space code, which to our knowledge has never been done before.

1.2 Structure

The conducted approaches are different for the two concerns. The approach for the first concern, i.e. *Memory access to user space must never be done directly, because the access may fault due to unmapped pages or insufficient access permissions*, is to extend all memory accesses with dynamic runtime checks, whereas the second concern, i.e. *Memory addresses must be checked for actually pointing to user space to prevent malevolent users from exploiting kernel drivers to access kernel space for them*, focuses on performing static analyzes to ensure the correctness of the code. Consequently, the document will be divided into two main chapters, where a background that concerns both chapters will be shared. The structure of this work will be the following:

Background introduces basic concepts and terms used in the following chapters.

- **Types in C language** introduces types in C language with focus on its aggregated types. This section is provided since it's the language most drivers are written in, thus some language knowledge is needed to understand the problematics treated in this work.
- **Operating system** introduces basic concept about drivers, system calls and safety and security issues related to pointers.
- **LLVM** introduces the LLVM project with focus on its intermediate representation, which is the representation the introduced analyzes will be performed on.

Train of Thought presents an overview of how this work countermeasure the different threats.

Dynamic runtime checks contains sections that describe the precautions performed to prevent the first concern, i.e. *Memory access to user space must never be done directly, because the access may fault due to unmapped pages or insufficient access permissions*.

- **Direct memory access to user space** describes how to prevent the concern from a theoretical aspect.
- **Implementations** describes how the precautions are implemented.
- **Measurements** presents measurements of the above mentioned implementations of these checks.
- **Result** presents the result from the conducted measurements.

Static analyzes contains sections that describes the precautions performed to prevent the second concern, i.e. *Memory addresses must be checked for actually pointing to user space to prevent malevolent users from exploiting kernel drivers to access kernel space for them.*

- **Accessing kernel space on behalf of user space** describes how to prevent the concern form a theoretical aspect.
- **Implementations** describes how the precautions are implemented.
- **Code examples** illustrates with code examples how the different extensions and analyzes work together to secure the drivers.

Discussions contains the evaluation of my work

Conclusions and Future work presents conclusions my work and propositions for future work.

Chapter 2
Background

This chapter will introduce basic concepts and terms used in the following chapters. First, section 2.1 will present the language in which most operating systems, including their device drivers, are written in, namely C, with focus on types. The following section 2.2 will present concepts of operating systems followed by the last section 2.3 which will introduce the LLVM project with focus on its intermediate representation.

2.1 Types in C language

This section introduces some aggregated types of the C language. Most of the types in C, such as `int` and `double`, are common to other programming languages. But there are some aggregated types that may be new to programmers using another language, such as Java. The following will provide a short introduction of these types, as well as a section about type conversions. A more detailed description can be found in [27].

2.1.1 Pointers

In C, it's possible to get the address where the data is stored. The type for holding an address is referred to as a pointer.

The operator & is used to access the address of data, which can be assigned to the pointer. That way, the pointer is pointing to that data. To later access that data the operator * is used, which is called the *dereferencing* or *indirection* operator.

```
int x = 1;
int y = 2;
int *z;          /* a pointer to an integer */

z = &x;          /* z is pointing the integer x */
y = *z;          /* y = 1 */
*z = 3;          /* x = 3 */
```

Above presents how to declare, assign and access the variable of a pointer. The first assignment takes the address of the variable *x*, and assign it to the pointer *z*. The situation after that assignment can be illustrated as in figure 2.1.

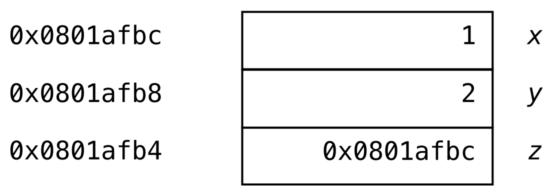


Figure 2.1: Pointer illustration where a pointer z is pointing variable x

The second assignment accesses the data where the pointer is pointing to. In this case, z is currently pointing to the address where the integer *x* is stored, which has the value 1. Therefore, *y* will be assigned value 1.

The third assignment accesses the data the pointer is pointing to, `int` *x*, and assigns it to 3.

This example only present pointers handling integer types. Pointers can also be used to hold addresses of all other types, such as `double`, `float`, functions and even pointers.

2.1.2 Structures

The aggregated type `struct` is used to hold types, which can be of the same- or of different types. The types within a structure is referred to as the members of the structure. A structure can be used as a convenient way to group variables that are related.

An example of such situation is if one wants to declare a two dimensional point. Desired attributes for such point are its x- and y coordinates. To make these two coordinates as one unit, instead of two separate variables, one could use a structure:

```c
struct point {
    int x;
    int y;
};
```

When later accessing the members of the structure, i.e. the x- and y-coordinate, there will be no doubt that they belong to the same point:

```c
struct point my_point;
my_point.x = 10;
my_point.y = 20;
```

A structure can also have members that is of the derived type `struct`, i.e. it can be nested. This feature could be used for example when creating a linked list.

2.1.3 Unions

A `union` can, such as a `struct`, be used to hold a set of variables which can be of the same- or different type. Unlike a `struct`, the members of the `union` can not be accessed simultaneously, since they occupy the same space. Only one member can be accessed at a time. Below present how a union can be declared in C:

```
union {
    char *string;
    void *pointer;
    int ival;
    float fval;
}  value;
```

The reason only one member can be accessed a time is because the memory area of a `union` is as large as its largest member. That means that all members of the `union` above have the offset zero from the base. Since it will be large enough to hold its largest member, it can be interpreted as all of its members.

A `union` can be used in situations where one needs to represent a value, which is of an unknown type. This can be useful for example when implementing a stack.

2.1.4 Type Conversions

The type assigned to a variable, can later be converted into another type. The general rule is that an automatic type conversion is allowed if the type conversion is to a new type that will not lose any information from the old type, such as when adding an `int` with a `double`, the integer will be converted to a double precision before the operation. That way, no information will be lost.

It is allowed to convert the other way, i.e. to convert a type into a "narrower" type. One example is if a `double` is converted to an `int`, then some precision of the data will be lost. Such situations may produce warnings.

Common situation to perform type conversion is when passing arguments to a function. The arguments of the function will automatically force a conversion for the passed arguments:

```
double sqrt(double d) { ... }
double x = sqrt(2); /* convert into double value 2.0 */
```

Conversions can also be forced with an operator called a cast. A cast is done by specifying the new type, surrounded in parenthesis, in front of an expression:

(new type) expression

2.2 Operating System concepts

This section will present operating system concepts. The first section presents the terminology used throughout the thesis followed by a description of virtual address space. Thereafter, device drivers as well as system calls will be presented briefly. The section about device drivers will present benefits and drawbacks of implementing a driver in kernel- or user space, with the intention of providing a description of why there are still drivers implemented in kernel space as well as why it's beneficial. Finally, the last sections will focus on security concerns when developing a kernel driver, especially concerning pointers. More detailed presentation of the topics can be found in several books, such as [1, 25].

2.2.1 Terminology

The terminology used in different operating systems is slightly different. To prevent confusion, this thesis will use the same terminology used within PikeOS, since that is the operating system used as a proof of concept in this work.

The Real-Time Operating System PikeOS enables safe execution of applications with different safety levels concurrently by providing different partitions. A partition is a set of resources (memory, CPU time and I/O access rights) where one or many applications can run within. An application is in PikeOS referred to as a process, which from the kernel point of view is a PikeOS task. Each PikeOS task is provided with its own set of virtual addresses within user space, and each task can contain a set of schedulable entities, referred to as threads, which all belongs to the same address space as the task.

Partition a defined set of resources (memory region, CPU time and I/O access rights) which is allocated statically at configuration time.

Task an entity allocated from its enclosing partition dynamically at runtime, where each specific task is provided with its own set of virtual addresses. All tasks within a partition share the available resources of that particular partition.

Thread a data structure for storing a full execution context. A thread belongs to a specific task, and shares the task's address space and resources with all threads that belong to that task.

Compared to Linux terminology a PikeOS task is roughly the same as a Linux process, and a PikeOS thread is roughly the same as a Linux thread.

2.2.2 Address spaces

The virtual address space is divided into two areas: user and kernel space. Kernel space is the set of addresses which holds the memory of the kernel. The code that runs in kernel space should then manage the different applications, each provided with its own set of virtual addresses, which run within user space. The kernel may access all memory, whereas the user task may only access its provided region.

The virtual addresses are typically 32 or 64 bits, depending on the architecture it runs on. A 32 bit address provides 4 GB addressable memory.

The division between the total amount of addresses depends on the operating system. As an example, the Linux kernel provides 3 GB to user- and 1 GB to kernel space as its default configuration for the x86 32-bit architecture. This division is often configurable, which means that there is no default separation used for all purposes. For example, another common approach is to divide equally 2 GB to user and kernel space. Also, the assignment of which space should be given the higher or lower addresses differs from various operating system, where three examples of common configurations are illustrated in figure 2.2.

Throughout this work *addresses* or *address* will be used as a synonym for *virtual addresses* or *virtual address*, see [25] for a description of different address types.

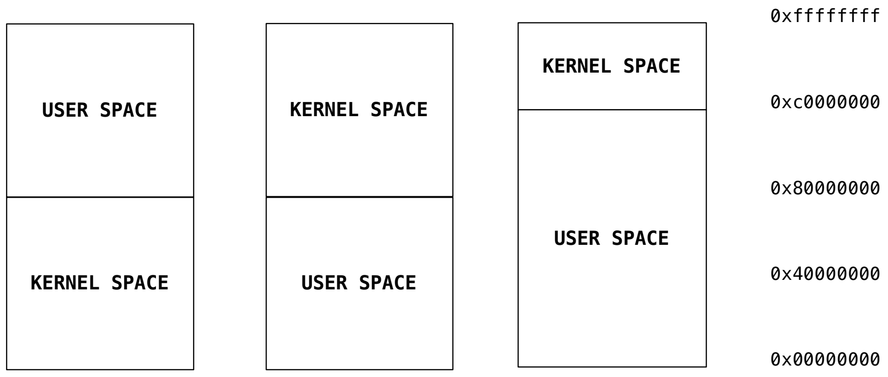


Figure 2.2: User- and kernel virtual address space

2.2.3 Device Drivers

Device Drivers are the system software which act as the interface between the operating system and the device. The connected devices can then be requested for their services from the kernel via I/O requests, such as sending/receiving network packages and reading/writing to disk.

An application which wants to request a service from a device will do so via a system call. The kernel will then be entered, and if the application has permissions, handle the communication to the device. The device may then return a result to the kernel, which will be sent back to the application. Device drivers may also be requested for their services from another device driver, where the two different ways to enter a driver is illustrated in figure 2.3.

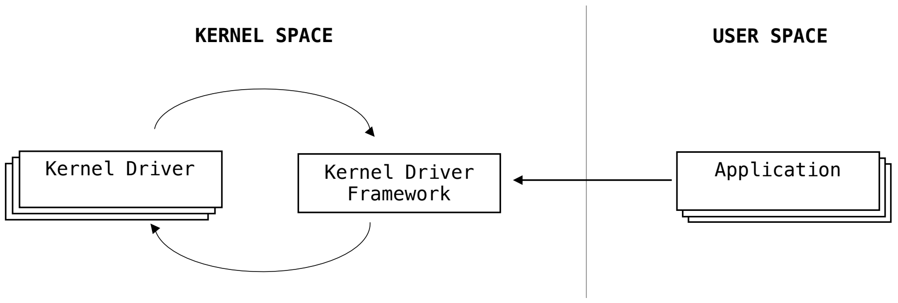


Figure 2.3: PikeOS kernel driver API

Most of today's operating systems offer a possibility to implement device drivers either in kernel- or in user space. Drivers implemented in kernel space will then be linked statically or dynamically with the kernel, depending on the specific operating system.

When writing a device driver that shall be linked with the kernel, the programmer needs to be careful and pay specific care. Since this introduced code will run in supervisor mode, there will be no security net, such as for accessing memory. Because of its criticality, it may be wise for some device code to be implemented in user space. Drivers implemented in user space are often referred to as user-space device drivers.

There are benefits and drawbacks to both user- and kernel space device drivers, which does not always make the decision of where to implement a driver trivial. Some of these aspects are presented below.

Advantages of kernel device drivers are:

Interrupts Handling interrupts with sufficiently low latency may only be possible in supervisor mode. Some actions like resetting an interrupt request bit in hardware may only be possible in supervisor mode.

Access Drivers can access memory as well as I/O ports directly.

Performance If the driver is implemented in user space, a request of a service will imply a context switch between the application issuing the request and the device driver who serves the request. This is needed for every communication to transfer data, whereas kernel device driver can manage the service directly.

Drawbacks of kernel device drivers are:

Safety and Security There are a lot of safety and security related issues because there is no strict hardware protection. This needs care, such as when performing memory accesses.

Debugging Debugging a kernel driver is not easy since it cannot easily, as for user space task, be executed under a debugger. Tracing the errors is also complex since they do not belong to a specific task. Further, an error may break the entire system, which could imply losing the evidence of why it occurred.

These are just some of the benefits and drawbacks for the different implementations. The typical dilemma is that you want to achieve the security benefits of implementing it in user space, as well as the performance benefits of implementing the driver in kernel space.

2.2.4 System calls

A task running in user mode can request services from the kernel via system calls. Because of this, developers of user applications do not have to study the different hardware devices to be able to use their services. The request only has to be according to the API. It also has security benefits, since the kernel can control the request before serving it. A brief presentation of some of the most common system calls follows:

read retrieves a given number of bytes from a device to a buffer. The number of bytes as well as a pointer to the buffer where data should be written to is provided as a parameter to the function.

write sends a given number of bytes to a device from a buffer. The number of bytes as well as a pointer to the buffer where data should be retrieved from is provided as a parameter to the function.

ioctl issue specific operations which can be customized to specific services offered by a device. This function is typically provided with a command to choose between the different functionalities, as well as a pointer which could be used to read or write data from/to.

The exact parameters and the types for the above presented system calls differ for various operating systems, and are also irrelevant in this context. The important thing with these is that they provide a way for a user to pass information from user- to kernel space via the parameters.

When a user application issues a system call, the request will enter the kernel where the mode is switched from user to kernel. The request will then be served in the same task context as the application issuing it. Since it will run in the same context, the user can provide a pointer to data within the application to the kernel. This data can then be accessed from the kernel, since the user virtual addresses are still represented by the task issuing the request. An simplified overview of how user space can enter data as parameters to system calls, which can later be accessed by the kernel is presented in figure 2.4.

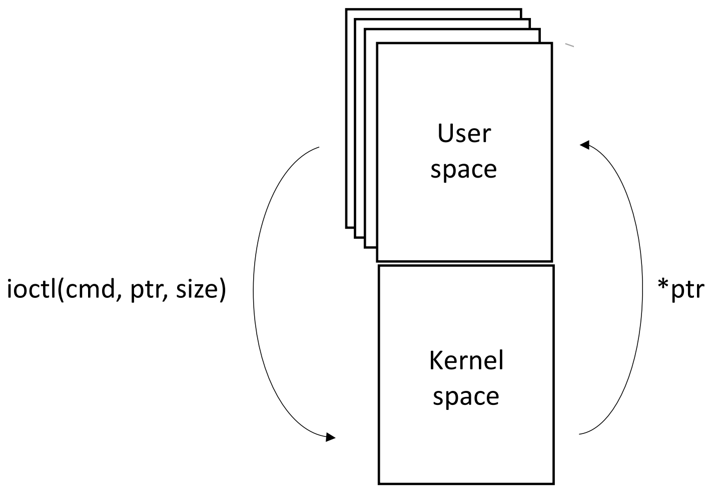


Figure 2.4: Overview of how user space can enter data as parameters to system calls, which can later be accessed by the kernel

2.2.5 Security issues

When writing a kernel driver, there are a lot of safety and security issues to think about. One is how to access pointers provided via a system call. Since the operations are handled in kernel mode, providing protection against common bugs from the kernel and the compiler is difficult, as stated in 2.2.3. There are the following concerns

Handling pointers Pointers must be handled with much care in kernel space. There are especially two types of pointers which must be distinguished, user- and kernel pointers, because the different pointers imply different safety measures.

Direct memory access Memory access to user space must never be done directly, because the access may fault due to unmapped pages or insufficient access permissions.

Accessing kernel space on behalf of user space Pointers passed in from user space must be checked for actually pointing to user space. A failure to do so may result in revealing or destroying memory that belongs to the kernel.

2.2.5.1 Handling pointers

Kernel drivers must handle data which comes from untrusted sources. This data must be handled with much care, since a single bug can introduce serious security vulnerabilities into the system. One common weakness is handling pointers which are passed from user space via system calls. Such system calls include read, write and ioctl, where ioctl is the most critical. The ioctl often copies additional data structures from user space, so that pointers may enter kernel space by an additional channel, supplementary to the system call. Therefore, ioctl is more error prone to mishandling pointers from user space.

Pointers handled in kernel drivers are typically distinguished into two types, user- and kernel pointers.

User pointer A pointer which comes from user space and should therefore be considered as unsafe.

Kernel pointer A pointer which is created in kernel mode and can thereby be considered as safe.

The importance to distinguish the two pointer types cannot be stressed enough, since the different types imply different safety measure. The distinction is not so clear, since both these pointers are declared in the same way. Since they cannot be distinguished by a simple look, it's very easy for developers to mix them. Also, in dedicated code reviews such bugs can be hard to find.

The safety measure for user pointers are especially important. These pointers are created in user space, and shall therefore only point to their restricted areas. They shall not be trusted, and should always be checked for their correctness prior to their use.

A pointer created in user space may be assigned an address in kernel space by a malevolent user. If this pointer later enters the kernel via a system call, a failure to check its correctness before dereferencing it may reveal or destroy information in kernel space. [14]

An additional problem is that kernel drivers often are implemented by third parties. Therefore, although using a safety critical operating system, introduction of kernel drivers with such bugs can break the entire system.

2.2.5.2 Direct memory access

When an application running in user space needs a service from some hardware, it will request the service by issuing a system call. The request will then be handled by a function running in kernel mode. This will be in the same context as the task requesting the service, i.e. the user address space will be mapped to the requesting task during the service of the system call. Since the operation is handled in kernel mode, it is able to access the whole memory area, i.e. both the area related to the user task as well as the kernel memory area.

Even though the kernel may access the whole memory, it shall not do so directly. Dereferencing user pointers may lead to unexpected memory faults due to unmapped pages or insufficient access permissions. There is no way to know how to handle an unexpected memory fault safely, hence most operating system will result in the kernel causing the system to panic. Therefore, all accesses to user space must be done through dedicated functions, which perform this in a controlled way.

One problem with this concern is that such bugs are very hard to find. The difference between dereferencing of a user- or kernel pointer cannot be distinguished directly by the eye, since both will be written exactly the same way, as explained in previous sections. The only way to distinguishing these two is to follow the pointer to gain knowledge about from where it may enter.

Drivers written in user space does not share the same problematic since they simply cannot access addresses they have no permission for. In the kernel however, such permission checks must be done manually if user space pointers are handled, hence, kernel drivers are more error prone.

A driver without the special precautions will seemingly work correctly, but silently have very serious bugs that the kernel is unable to detect. Also, since unmapping and remapping are time critical, if the poor code crashes due to map accesses, it is highly likely to be in corner case situations that are very difficult to debug. This may lead to low-quality code that is assumed to be correct.

2.2.5.3 Dedicated access functions

The kernel may access user space addresses, but should do so with much care. There are two dedicated access functions in PikeOS, drv_memcpy_in and drv_memcpy_out that should always be used for this purpose. The first one copies a given number of bytes from a location which may potentially be located in user space to a location in kernel space, and the second copies a given number of bytes from kernel space to a location which may potentially be located in user space.

- drv_memcpy_in(dst, src, size)
- drv_memcpy_out(dst, src, size)

Both functions are architecture dependent and quite similar, hence the following will only explain the drv_memcpy_in in detail, where a simplified control flow is presented in figure 2.5. The following description will expect the source to be located in user space.

Before invoking the given function, one must first ensure that the source, *src*, belongs to user space, if that is the expected situation. A verification of the source must be performed prior to this function, otherwise safety and security vulnerabilities will be introduced.

After performing the proper checks to ensure the correctness of the source location drv_memcpy_in can be entered. This function will invoke yet another function to ensure that the destination, *dst*, is located in kernel space. When the correctness of the destination is ensured, an architecture dependent function will be entered where the actual work will be done.

In case the architecture dependent function fails, an error code will be generated. When this operation is performed, the error is known and can thereby be handled in a controlled manner. Since PikeOS is preemptive also in the kernel, there is no check prior to the access that can guarantee that it will be fault-free, because the mappings may change in between the check and the actual access. Thereby, this results in an implementation which will perform the access, and in case of failing result in a state where the situation is handled safely.

If these functions are not used and the access fails, the reason will be unknown. Since this will then be an unexpected memory fault, there is no way to handle such situation safely, hence the PikeOS will cause the system to panic. Therefore, pointers potentially pointing to user space must be accessed via these dedicated functions.

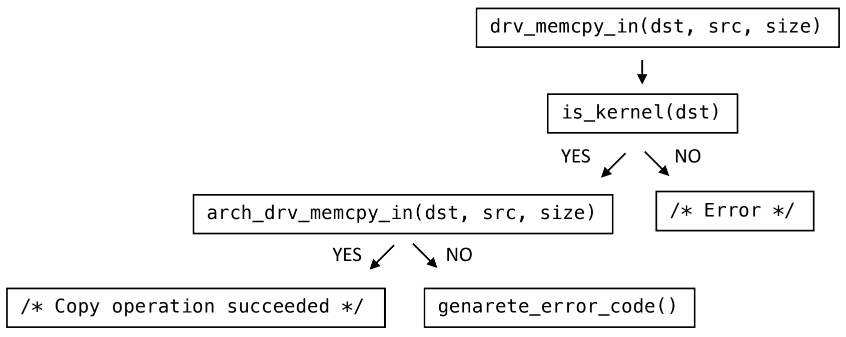

drv_memcpy_in(dst, src, size)
is_kernel(dst)
YES
NO
arch_drv_memcpy_in(dst, src, size)
/* Error */
YES
NO
/* Copy operation succeeded */
genarete_error_code()

Figure 2.5: Simplified call graph for the PikeOS function drv_memcpy_in

There are similar functions in other operating systems, for example Linux use copy_from_user and copy_to_user which should also be used when performing access to user space [15].

2.2.5.4 Accessing kernel space on behalf of user space

Programmers writing kernel drivers must be careful when using data provided from user space. One particular consideration is that pointers passed from user space must be checked for actually pointing to user space.

Since kernel drivers may request services from other drivers, the check is even more complex. Namely, these functions may be entered by another kernel driver, where data is considered safe, or via a system call where data is provided from user space and should therefore be considered as unsafe data.

The unsafe data should be checked for actually pointing to user space whereas the safe data would not need to be checked. This makes a check of the data complex, since you need to know from where the data was provided from to be able to perform the check in a correct way.

One function, commonly implemented in drivers, that takes pointers as its parameters is the read function. This function is provided with a pointer to a buffer and the number of bytes it's supposed to read. Therefore, it is critical to check that the pointer is pointing to a buffer that resides in user space. These checks are in PikeOS performed by the kernel driver framework, which is the first entering point in the kernel when a system call is issued. This means that when the pointer is verified by the framework, the requested driver will be invoked. That driver can thereafter use this pointer safely, i.e. since it's verified, as a user pointer.

Another situation when the check gets even more complicated is when the data is of an aggregated type, such as an `array`, `struct` or `union`. When using such types, the members will also need to be checked for actually pointing to the right memory area.

Aggregated types are often used in the system call ioctl. Ioctl may contain different functionalities, where the different functionalities will be given different numbers. Thereby, the user can choose which functionality is desired, by providing its number as one of the parameters. This means the pointer that is passed in to an I/O control may be interpreted as different types, depending on the desired service. Therefore, there is no easy way to perform a check that fits all these services, except in the driver itself.

Lets assume a function of a kernel driver that takes a pointer to a structure as a parameter, where this function is then entered via a system call where the user provides a pointer to the data. The structure will then need to be checked so that it belongs to user space, which is in PikeOS done by the kernel driver framework. If the structure passes that check, then it's safe to copy the structure into kernel space.

Now the structure belongs to kernel space, were there are higher privileges. If one of the structures members is a pointer, it may point to kernel space. Since it is operating in kernel mode there will be no safety net, i.e. that access is now allowed. Although the access is allowed, it's requested from user space and must therefore be prevented. This can cause huge problems and therefore it's important that also the members of an aggregated type are checked, see figure 2.6.

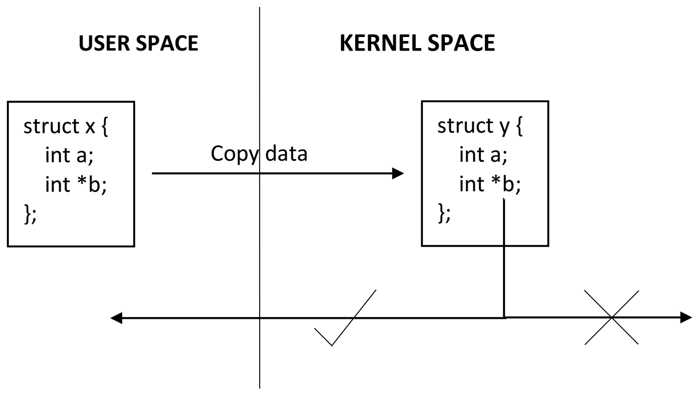


Figure 2.6: Copy data from user- to kernel space

The situation can be even more complex. Imagine that the structure contains a structure itself, where that structure contains an array of pointers. Then the pointers that belong to the array will also need to be checked at runtime for actually pointing to user space prior to their use.

2.3 LLVM

This section will present an overview of the LLVM project with focus on its intermediate representation. This work will compile kernel drivers into the LLVM intermediate representation, where this representation will be used to conduct analyzes and code transformations in order to make the drivers more secure. The drivers will then be compiled into ELF object files and linked with the PikeOS kernel.

2.3.1 LLVM Overview

LLVM is a C/C++/Objective-C compiler which started as a research project by Chris Lattner. The project is open source and today widely used for creating sub-projects in both scientific research contexts as well as in industry. An overview of the projects different components is illustrated in figure 2.7.

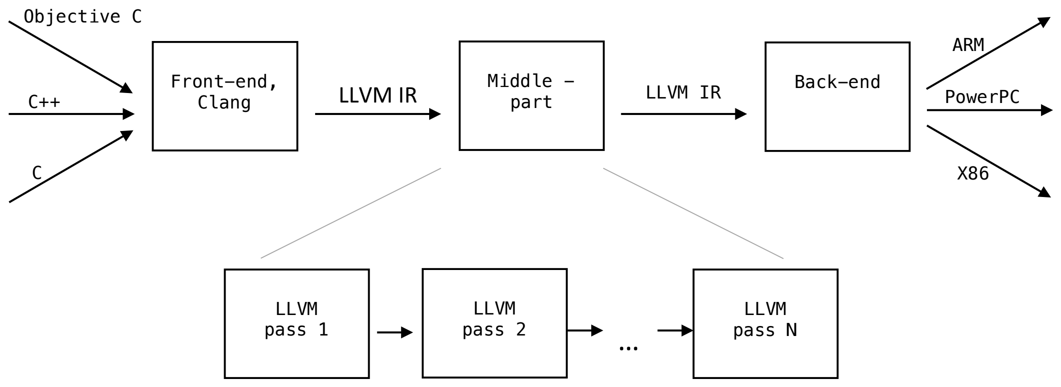


Figure 2.7: The LLVM compiler steps

The project can be divided into three parts, the front-end, the middle-part and the back-end. LLVMs front-end is called Clang and it takes the source code as input and produces the LLVM Intermediate Representation (IR) as output. This part is dependent on the language of the source code, e.g. C, C++ or Objective C.

The Intermediate Representation is then handled by the middle-part. The middle-part will use the same language for its input and output. This part is independent of the language of the source code as well as of the machine it will run on. The input can be transformed with so-called LLVM Passes, which can be several or none. For example, when enabling optimization in Clang, such as with the -O3 flag, the code will run several passes to perform the optimization. While if no optimization is enable, then it will not run through any of these LLVM passes. The order of the passes depends on the order one declare its flag, i.e. -pass1 -pass2 will first run the LLVM pass related to "pass1" and thereafter the pass related to "pass2".

The LLVM IR will then be the input to the back-end which will generate assembly code as its output. This part is dependent on the machine running the code. In this work, we will use LLVM version 3.6.

2.3.1.1 Debug Information

The LLVM front-end Clang is able to produce debug information. The debug information is a way to provide information of what the compiler optimized away or restructured. For example, a function that is marked as an inline function, may be inlined in the front-end.

Since the `inline` keyword is only a suggestion to the compiler, there is no guarantee that it will indeed be inlined. But if it were, then by enabling debug information, one can trace that this code was inlined. Such information can be very valuable when conducting analyzes.

The generated debug information in LLVM IR will be attached to the instruction it belongs to. In the intermediate representation will this be indicated by the string "!dbg" followed by the location to the debug information in the end of the instruction. Typical information tagged to an instruction is the name of the file it belongs to and the line in the source code that it's related to.

There are different levels of which/how much information should be generated by Clang, such as no debug information, only information about the line number, all information supported by the front-end, etc. In this work, the debug information will be used to enable useful debug messages for the static analyzes part, where we will request generating all debug information supported by Clang.

2.3.2 LLVM Intermediate Representation

There are three different forms of the LLVM Intermediate Representation, each suitable for different situations. One of these is the human readable representation, which is the representation we focus on throughout the thesis.

In LLVM, the structure of a program will be divided into functions, basic blocks and instructions.

2.3.2.1 Program, Functions and Instructions

A program contains a finite set of functions where exactly one is the entry point of the program. The functions of a program are defined and/or called within the program. All functions contain a sequence of LLVM instructions. These instructions are classified into different subclasses; memory instructions, terminator instructions, binary instructions, bitwise binary instructions and others. Further description of the instructions can be found in the LLVM Language Reference Manual [19].

2.3.2.2 Basic block

The control flow of a function divides its instructions into a set of basic blocks, where the instructions within a basic block are ordered into a straight-line sequence. A basic block has:

- exactly one entry point
- maximally one exit point

This means that there is no way to enter a basic block except from the first instruction of the block or exit at any instruction except the last one, which is referred to as the terminating instruction. Also, when the first instruction is executed, then the following instructions will also be executed in their respective order.

2.3.2.3 C example

The following C code will illustrate the program structure generated in LLVM:

```c
int main(void)
{
    int i;

    for(i = 0; i <= 10; i++)
       printf("i: %d\n", i);

    return 0;
}
```

In the LLVM human readable IR, this will be represented as:

```
@.str = private unnamed_addr constant [7 x i8] c"i: %d\0A\00",
                                                        align 1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %i = alloca i32, align 4
    store i32 0, i32* %1
    store i32 0, i32* %i, align 4
    br label %2

; <label>:2                 ; preds = %8, %0
    %3 = load i32* %i, align 4
    %4 = icmp sle i32 %3, 10
    br i1 %4, label %5, label %11

; <label>:5                 ; preds = %2
    %6 = load i32* %i, align 4
    %6 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds
        ([7 x i8]* @.str, i32 0, i32 0), i32 %6)
    br label %8

; <label>:8                 ; preds = %5
    %9 = load i32* %i, align 4
    %10 = add nsw i32 %9, 1
    store i32 %10, i32* %i, align 4
    br label %2

; <label>:11                 ; preds = %2
    return i32 0
}

declare i32 @printf(i8*, ...) #1
```

In this example, the module contains one function, i.e. the main function. The function contains five basic blocks, where each basic block is identified with a label and contains exactly one terminating instruction.

The basic blocks contain one or several instructions. The first basic block, which is provided when calling the main function, contains instructions that allocate space for the return value, in this example denoted as "%1", and the integer *i*, that later will be used in the for-loop. Note how the local variable *i* is identified with the beginning of a "percentage" sign, % and the global data with the beginning of an "at" sign, @. This basic block ends with a branch instruction to the basic block identified as "<label>:2".

Basic block "<label>:2" will load the value of *i*, and compare if it's less or equal than 10. Depending on the result of the comparison, it will cause control flow to either the basic block "<label>:5" or "<label>:11".

Basic block "<label>:5" will load the value of *i*. This value will then be used as a parameter, combined with the string "i: %d\n" that is identified as @.str, to the printf function. The printf function is declared, with its return type as well as its parameters types. The block ends with a branch instruction causing control flow to basic block "<label>:8".

Basic block "<label>:8" loads the value of *i*, add one to it, and store the new value in the space allocated for variable *i*. The block ends with a branch instruction causing control flow to the basic block "<label>:2".

Basic block "<label>:11" will simply return the return value for the function main, in this example 0. As can be seen from this example, LLVM is Static Single Assignment (SSA) which means that a variable is assigned exactly once.

Program analyzes can be performed on the generated intermediate representation. One area of analyzes is optimizations which typically reduce the number of instructions or structure/change them in a more efficient way. Another area of program analyzes is to ensure the correctness of the code. These analyzes are typically based either on functions or on the whole program. The work in this report focuses on improving the correctness of a program, where the analyzes and code transformations will be function based.

2.3.2.4 Aggregate Types

Aggregate Types are types that can hold one or several types, such as `struct`, `union` and `array`.

An `array` hold a sequence of variables, referred to as its members, which are of the same type. In LLVM IR is an array represented as

[<number of members> x <type>]

For example, [7 x i8] is an `array` with 7 members of 8-bit integer type.

A `struct` have members which can be of different types. In LLVM IR is a `struct` represented as

{ <list of member types> }

For example, { i32, float, i32 } is a `struct` with two 32-bit integer values and one floating point value as its members.

LLVM IR 3.6 does not support `union` types as one of its aggregated types. The only aggregated types supported is vectors, structures and arrays. But, not to be confused, the whole C language is supported and can be translated into the LLVM intermediate representation. For example:

```c
union Foo {
    int a;
    int *b;
    double c;
};
```

Becomes in LLVM IR version 3.6

```
%union.Foo = type { double }
```

That means that it's fully supported in terms of generating the intermediate representation from source code which includes `union`, but when looking at the generated intermediate representation, data is lost. Only one member of the `union` is presented as a visible type, and, when changing type from one of the members into another, simply a cast between the two member types will be presented in the intermediate representation. Also, the type in the LLVM IR language will not be a `union` type, because there is no such type in the language. Instead, it will be created as a `struct` type with only one of its members. But, by looking at the generated characters, it could be distinguished if the structure type is generated from a `union`, i.e. a `union` will in the LLVM IR be a `struct` type which is declared "%union.Foo", whereas a `struct` will in the LLVM IR be a `struct` type which is declared "%struct.Foo", see table 2.1.

C	LLVM IR Type	visible indication	visible members
union	struct	%union.<name>	1
struct	struct	%struct.<name>	all

Table 2.1: Generated LLVM IR information from `struct` and `union` types

Further, it's also not possible to insert LLVM IR instructions to create a `union` type, because of the same reason, there is simply no such type. Therefore, if that is desired, one have to create a structure that contain one of the members, and insert cast instructions whenever the type needs to be changed.

2.3.2.5 Address space

The LLVM type system provides an optional address space for pointers, which can be provided as an attribute. This attribute will precede the other attributes that are assigned. The meaning of the address space is target-specific, where the default address space for all pointers is address space 0.

A pointer can be provided with an address space in the following way

```
void __attribute__((address_space(N))) *a;
```

This will assign the attribute address space N, where N is an integer, to the pointer *a*. This attribute will be presented in the LLVM Intermediate Representation:

```
%a = alloca i8 addrspace(N)*
```

The address space attribute only belongs to the pointer it is assigned to, i.e. in case of assigning an address space to a pointer that points to an aggregate type, all members of pointer type will still be assigned the default address space 0. The address space of the members can be changed in the same way the aggregated type was assigned its attribute.

Chapter 3
Train of Thoughts

In the following, the problems as well as the countermeasures will be presented to provide an overview of the work. This will be illustrated with a flow chart, see figure 3.1, together with a description. The purpose of this chapter is to provide a guidance of the different parts and how they work together to provide a solid solution.

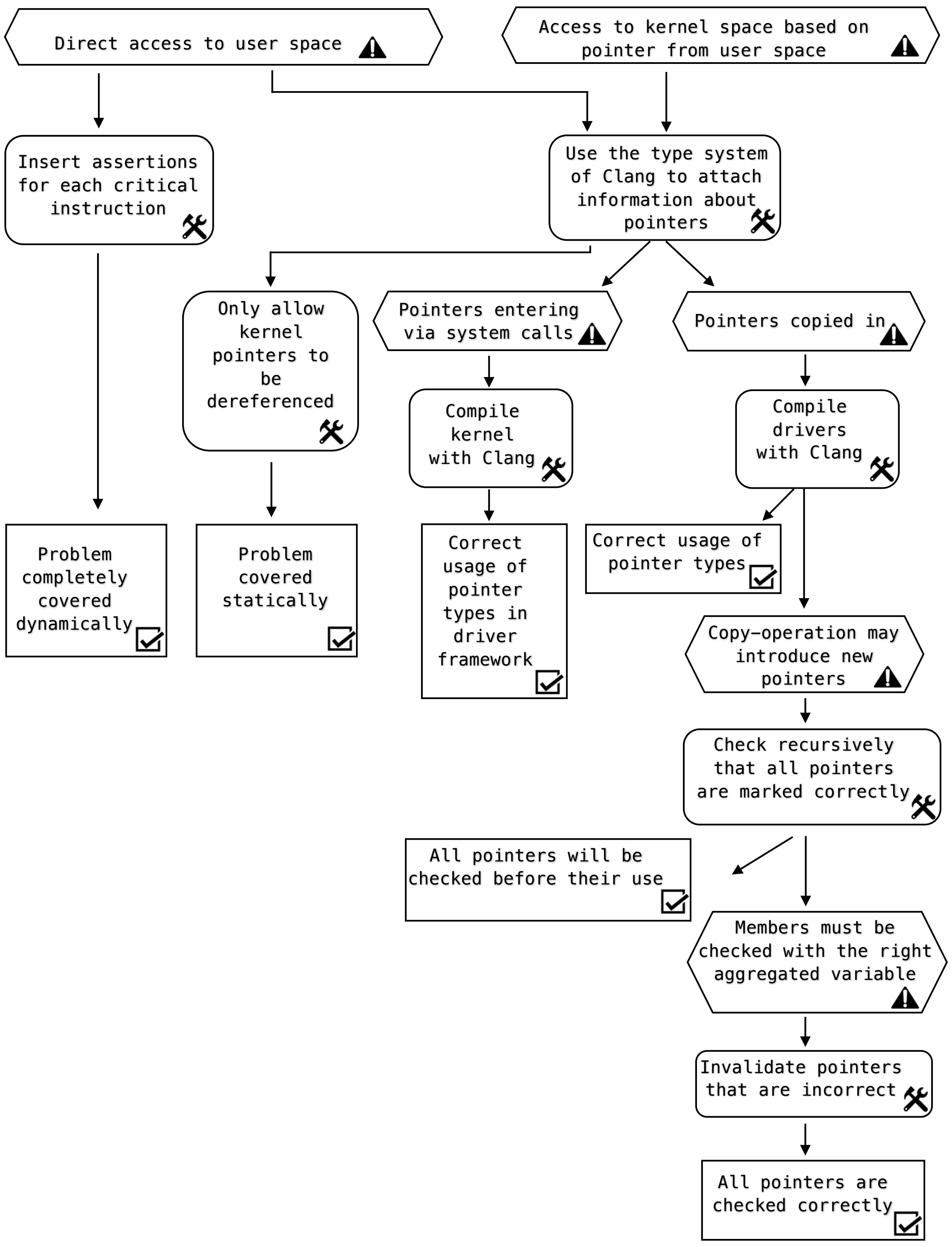

Direct access to user space
Access to kernel space based on pointer from user space
Insert assertions for each critical instruction
Use the type system of Clang to attach information about pointers
Only allow kernel pointers to be dereferenced
Pointers entering via system calls
Pointers copied in
Compile kernel with Clang
Compile drivers with Clang
Problem completely covered dynamically
Problem covered statically
Correct usage of pointer types in driver framework
Correct usage of pointer types
Copy-operation may introduce new pointers
Check recursively that all pointers are marked correctly
All pointers will be checked before their use
Members must be checked with the right aggregated variable
Invalidate pointers that are incorrect
All pointers are checked correctly

Figure 3.1: This works main concerns with a flow chart of the founded threats and the proposed countermeasures

Direct access to user space: One of the two major problems treated in this work is that direct dereferences of pointers to user space should in all situations be prevented.

One of the approaches to prevent this problem is to extend kernel drivers with dynamic runtime-checks. This is enabled by compiling the drivers into the LLVM intermediate representation where the robustness assertions will be inserted prior to all critical instructions 4.1. The very few instructions to perform an access to memory, also compared to the RISC instruction set [6], enables complete coverage against this concern.

Access to kernel space based on pointer from user space: The second major problem treated in this work is to prevent users from exploiting kernel drivers to access kernel space for them. Because of the easy way to do memory access in C with a combination of running in supervisor mode where there is no safety net, these bugs are unfortunately easy to introduce into the system.

This security vulnerability starts with the system calls that allows the user to enter data from user- to kernel space via pointers, such as read and ioctl 2.2.4. These pointers must point to their restricted area, and must be checked for that fact before using them. Since drivers can also be requested for their services by other drivers, these functions may as well be entered from kernel space, where the entering pointers may point to kernel space in a correct manner.

The problem is two fold:

- Make it easier for the developer to distinguish which data has been checked, and which has not.

- Make it easier for the developer to distinguish if a pointer belongs to user or kernel space.

This problem can be prevented with the use of the compiler Clangs type system. The front-end Clang provides a way for the developer to introduce new types which can be attached to pointers by assigning them to different address spaces 2.3.2.5.

By introducing new types, Clang will automatically help the developer to separate the usage of a specific type for a specific situation, i.e. prevent one from mixing them up. Further, these types are also a visible indication if a pointer is checked/unchecked or a user/kernel pointer 5.1.1.

This does not provide a watertight solution. Especially since it puts a lot of responsibilities on the developer to provide casts between these new types in a correct manner. Correct in a sense that a type is only changed when the pointer indeed corresponds to the new type, i.e. casting an unchecked user pointer to a checked user pointer is only done if it has been checked. By introducing new functions which perform checks as well as providing a new type to the pointer, this would be handled correctly 5.1.2. Although, the introduction of the new functions does not prevent a developer from ignoring them and simply perform their own casts.

Another benefit by differentiate the pointers by assigning them to different address spaces is that this attribute will be visible also in the LLVM intermediate representation 2.3.2.5. Thereby, analyzes can be inserted in form of LLVM passes to enable a correct usage of the types, i.e. to be able to verify the correctness of the program.

In this situation, a verification that casts between two pointer types is only performed in the introduced dedicated functions is necessary 5.1.3.1. By enabling such analyze, one can be sure that all pointers, initially marked with the right type, will be checked before being assigned a new type. The transformation from the old type to the new type then enables that the pointer can be used in another context, as approved.

Pointers copied in: Since this relies on that pointers are initially provided with the right address space, it is critical to differentiate which pointers should initially be provided with what address space. By extending the API for kernel drivers with the new types, the developers would simply have to follow this to get it right 5.2.1. Also other parts of the kernel would need to be extended, such as performing checks with the new functions in the driver framework as well as providing the extended types to some service functions, such as the critical copy-function 2.2.5.3. With these extensions, a compilation with Clang of the kernel would then verify that pointers are used correctly also in the driver framework, which in fact is as a bonus of this extension.

Pointer entering via system calls: Pointers which have been verified for their correctness by the framework will then enter the drivers. These entering pointers are still user pointers, and should thus not be mistaken for kernel pointers. By simply following the extended API, all entering pointers will be assigned the right type and a compilation of the driver will thereby provide a correct usage of the introduced pointer types, e.g. prevent one from mixing the different pointer types.

Copy-operating may introduce new pointers: Since the framework has verified the correctness of the pointers, it is safe to copy their data from user to kernel space. A copy operation of this kind may introduce new pointers, entering as members of aggregated types. These introduced pointers have not been checked for their correctness and should thereby not be trusted. They are neither covered by the extension of the API to be ensured to have been assigned the right type, nor controlled by the framework, hence these must be checked. By providing a static check that can verify that all data copied in is marked with the right type, it can be stated that all pointers will be checked before their use 5.1.3.2.

Members must be checked with the right aggregated variable: To check a member of an aggregated type, information about where it entered from must be known. An aggregated type which enters from kernel space with an invocation from another driver may contain pointers which points to kernel space in a correct manner. On the contrary, data entered from user space via system calls must only contain pointers that point to their restricted area. To be able to decide whether the members may point to kernel space or not, the aggregated variable must be provided. This is a critical point of the construction, since the check is based on the aggregated variable, thus an incorrect aggregated variable is enough for an introduction of a safety and security whole in the system.

To prevent this from occur, an additional feature is provided. This feature checks all member data, that has not yet been checked, directly when it is copied in. If some of these members are incorrectly pointing to the wrong area, an insertion of an assignment to an invalid address will be provided. That way, in case these pointers are later accessed, it will fail, and if not, there will be no complications 5.1.3.4. This extension will enable a kernel driver where all entered data is checked correctly.

Direct access to user space: With the introduction of differentiating user from kernel pointers, another analyze was extended to provide a static approach to prevent direct accesses to user space 5.1.3.5. This analyze will compare to the dynamic approach enables a faster technique identifying the critical parts of the code.

Chapter 4

Dynamic runtime checks

This chapter will present sections related to the first concern, i.e. *memory access to user space must never be done directly, because the access may fault due to unmapped pages or insufficient access permissions.* The first section 4.1 presents the precautions from a theoretical aspect. The precautions will be implemented in two different ways, both presented in section 4.2. Last, the measurements conducted on the different implementations will be presented in section 4.3 with their result in section 4.4

4.1 Direct memory access to user space

Handling pointers in kernel drivers must be done with care, especially considering the separation between user and kernel pointers. When dealing with user pointers one must consider them as untrusted and should always check their validity. Even after correct checks are performed on user pointers to ensure their validity, it's crucial to use dedicated access functions to access their data, see section 2.2.5.3.

To ensure that all accesses to user space will be through the dedicated access functions, assertions prior to all memory accesses can be inserted. These assertions will then perform checks which will ensure that the memory access will only be performed in case it's about to perform an access to kernel space.

The LLVM Intermediate representation can be exploited for this purpose. The restricted number of instructions which can perform a memory access, enables a sound check. If this would be performed on the C source code, one would need to parse the code and search for sequences which performs accesses. Since you then must treat the whole C lanugage, the possibility to miss some corner case situations is more likely.

LLVM IR provides seven memory access and addressing instructions; `alloca`, `load`, `store`, `fence`, `cmpxchg`, `atomicrmw` and `getelementptr`. Of these instructions, `cmpxchg`, `atomicrmw`, `load` and `store` will perform memory access. The first two instructions are used for atomic operations, which is operations where the processor can read and write in the same bus operation, i.e. either both operations or non of them will be performed. Instruction `cmpxchg` loads a value A from a memory address, compares it with a value B, and stores back another value C if A and B are equal. `Atomicrmw`, loads a value and stores back the sum of that value combined with another. The third and the fourth memory access instructions, `load` and `store`, which as their name indicate either load or store a value from/to memory and those are the instructions that will be generated in most situations. Further information about these instructions can be found in LLVM Language Reference Manual [19].

Aside from these operations, a direct access to user space can also occur via a function pointer. To distinguish accessing functions in user space, one needs to perform checks prior to instructions which will perform calls to other functions. In LLVM, this is done with the call and invoke instructions, where we will only focus on the call instruction since invoke is not used for C.

Prior to these instructions, a check should be inserted, which shall ensure that the access is allowed. To be able to check if the access is allowed, one needs to check if the address about to be accessed belongs to kernel or user space. If the memory is an address that is within user space, the execution shall not continue, preventing the access from actually occurring, which in this work will be performed by raising an assert-failure. In contrast to that, if the check provides information that it's about to access an address within kernel space, the execution may proceed.

By inserting these checks which will prevent the execution in case its about to access user space, critical points in the source code will be found during runtime. These should then be changed into proper accesses via dedicated functions, making the source code more secure.

There are especially two things needed to perform the check:

- The address which needs to be checked
- Which addresses belong to which address space

These critical instructions will also be generated when reading or writing to global and local variables. These checks always will always result in approving that the address relies in kernel space, therefore to reduce the number of inserted checks one could exclude checking accesses to values which are local or global without jeopardizing the soundness of the check.

4.2 Implementations

4.2.1 Call external function

One approach to implement the check which should be inserted prior to all critical instructions is to insert a call to an external function. This function will then perform the actual check as well as preventing the execution in case it's about to access user space. We implemented the check function in an external source code file, written in C.

The first information needed to be able to perform the check is the address which is about to be accessed. This information was retrieved from the critical instructions performing the access, since all instructions which may perform an access to user space hold information about which address it's about to access.

By implementing the check in C, information from other sources could be retrieved. This is especially advantageously to retrieve the second information needed to be able to perform the check, namely which addresses belong to which address space. To check if an address belongs to user or kernel space is most likely implemented within the OS, where this approach use PikeOS address space check functions. These functions will in PikeOS "adapt" to the current configuration, i.e. if its 32- or 64-bit and how the addresses are divided to user- and kernel-space.

The performed check as well as the prevention of execution in case of accessing user space was implemented in four ways. The check as well as the prevention of the execution was implemented in two ways.

Both implementations, which perform the control, took advantage of known information from other source files. The first one check if the address is in user space and the second if the address is within kernel space. Both of these approaches need the same external information to be able to perform the check, namely the address it should perform the check on. When inserting a call to the check-function in the LLVM IR, this information was retrieved from the critical instruction and entered the check-function as a parameter. Since LLVM is very strict about types, a cast to the proper parameter type is also needed. This results in two inserted instructions per check, a cast- and a call instruction.

If the check shows that it's about to perform an access to user space, the execution shall not proceed. This was implemented by raising an assert failure, in two different ways:

```c
#define assert(x) ((x) ? ((void)0) : assert_failure(__FUNCTION__,
                                     __FILE__,
                                     __LINE__,
                                     ""))

void assert_failure(char *function,
                    char *file,
                    int line,
                    char *message);
```

The first implementation raises an assert failure by invoking the function via macro expansion. The assert failure function takes four parameters: the function, file, line and a message. When invoking it via the macro expansion, the first parameters is a defined macro within PikeOS, and the following two will be provided by predefined macros. This is a macro used to debug within PikeOS, there the reason that the assertion fails can vary widely. Therefor, there will be no informative error message. The result will be an uninformative message to the user. Also, the provided information about the function and line will lead to the check-function, which is of course true, since that is indeed the place where the assert failure was raised. But, the reason for raising the failure will have to be traced back to the instruction after it was invoked. This can be somehow misleading, especially if you are using these insertions for the first time.

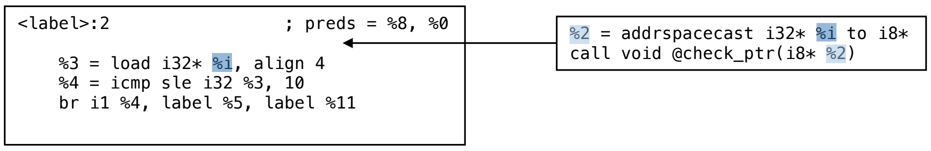
The other approach will give an informative message to the user, as well as valuable tracing information about the actual error. When inserting a check, the instruction that is about to be checked for its access is known. By enabling debug information, data will be attached to this instruction. This data was then used to obtain information about which line and file this instruction was generated from. The intermediate representation also holds information about which function the instruction belongs to. Thereby, this information can be passed to the check-function along with a valuable message. This led to an informative error message, containing the typical information you need, the file, function and the line number where the critical point is with a valuable message informing why the error occurred.

The number of inserted instructions varies with which error generating approach is preferred. The first approach needs no extra information from the intermediate representation, since it will basically raise an assert failure and retrieve information via predefined macros. The second approach needs four additional parameters from the LLVM IR. When a module runs through the created LLVM pass, the error message as well as the file name will be created once as global variables. The function and the line number will change throughout the pass, hence these needs to be created for the specific instruction which is currently checked.

An overview of the different approaches is presented below:

```
/* Approach 1 */
if(is_user(address)) {
    assert_failure(function, file, line, message);
}

/* Approach 2 */
if(is_user(address)) {
    assert(0);
}

/* Approach 3 */
if(!is_kernel(address)) {
    assert_failure(function, file, line, message);
}

/* Approach 4 */
if(!is_kernel(address)) {
    assert(0);
}
```

The situation when inserting a call to the external function which follows approach 2 can be illustrated according to figure 4.1. This figure illustrates a kernel driver, which have been compiled into LLVM intermediate representation. The generated code is thereafter used to search for critical instructions, which in this case was found, i.e. the load instruction. To ensure that this is not a direct access to user space, the address that is about to be accessed is checked by inserting a call to an external function, in this case called *check_ptr*.



```
<label>:2                    ; preds = %8, %0

    %3 = load i32* %1, align 4
    %4 = icmp sle i32 %3, 10
    br i1 %4, label %5, label %11
```

```
%2 = addrspacecast i32* %1 to i8*
call void @check_ptr(i8* %2)
```

Figure 4.1: Inserting LLVM instructions which will generate a call to an external function, which in return will ensure that there are no direct accesses to user space.

4.2.2 Inline instructions

The check prior to memory access instructions was also implemented by inlining LLVM instructions instead of inserting a call to an external function. The inserted LLVM instructions is both performing the check as well as preventing execution in case the instruction is about to perform an access to user space. The idea with this approach was to implement a check which was more efficient than the previous approach.

The previous approach inserted a call to an external function, where the actual job was performed. This approach will instead insert similar LLVM IR instructions to the ones generated from the separate source file. That way, instructions will be reduced, like the call to the external function.

The check was implemented for a 32-bit architecture that divides the total amount of addresses equally between the two spaces, which is a common configuration. By implementing a check for a configuration where the addresses are divided equally between the two spaces, the address can naively be compared if its less than the middle address value. The result from that comparison can then be used to decide whether to continue the execution or not. This will result in three inserted LLVM IR instructions, one to perform a cast to the proper type, one comparison instruction and a branch instruction which depends on the result from the comparison instruction.

Since this insertion contains a terminating instruction, i.e. the branch instruction, the basic block where the check is inserted will need to be separated into two blocks. One block will then contain the instructions until the memory access, referred to as block A. The rest of the block will then be moved to a new block, which also contains the memory access instruction, referred to as block B. The situation after the separation of the block is illustrated in figure 4.2.

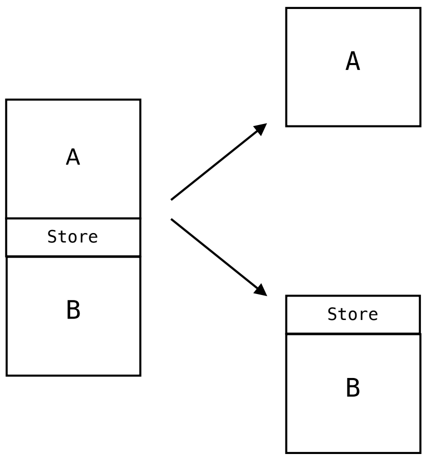


Figure 4.2: Creating Basic Block A and B

The instructions which will perform the check will then be inserted at the end of block A. The terminating instruction of this block will be the branch instruction which depends on the comparison. Depending on the result, the execution will either continue, i.e. at block B, or the execution will be prevented by causing control flow to another block, referred to as the error block. The situation after inserting the instructions is illustrated in figure 4.3. Both figures illustrate the situation of a store instruction, but the same procedure goes for the other memory access instructions.

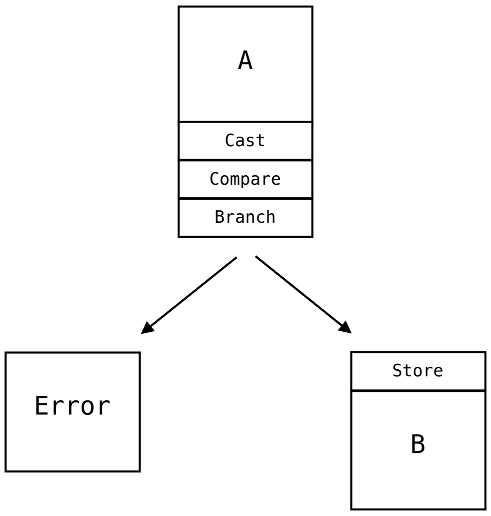


Figure 4.3: The final created block situation

The error block prevents further execution by issuing an assert failure, described in section 4.2.1 as the informative error, which takes four parameters, the file, function, line and a message.

The filename as well as the message is created when a file is entered in the created LLVM pass as global variables, since it's common to all functions. The function needs to be created for each function analyzed, and the line number for each line where a check is inserted. Since these parameters depend on the current instructions which need to be checked, the call instruction cannot be generic to all purposes. This means that since the parameters to the function raising an assert failure are different, there has to be a new error block created for all checks.

Below present an overview of the created LLVM IR code for two error blocks. The first three global variables, the filename, error message and function, will be used to generate an informative error message. The first error block, referred to as "error_block" in the example, will use all these variables when invoking the assert_fail function followed by an instruction that can not be reached, since the assert_failure function will terminate the program. The following error block, referred to as "error_block3", will also use these three variables. The difference between these two is the third parameter, namely the line number. The first error block has the line number 8 whereas the second error block has number 10 as its parameter. Also, since the code transformation is function based, the function may vary between the different error blocks. Therefore, there have to be new blocks created for every possible function and line number where an error can occur.

```
@filename = private unnamed_addr constant [12 x i8]
                            c"llvm_test.c\00"
@error_msg = private constant [28 x i8]
                            c"Direct access to user space\00"
@function = private unnamed_addr constant [5 x i8] c"main\00"

error_block:            ; preds = %0
  call void (i8*, i8*, i32, i8*, ...)*
    @assert_fail(i8* getelementptr inbounds ([5 x i8]* @function,
                                            i32 0,
                                            i32 0),
                 i8* getelementptr inbounds ([12 x i8]* @filename,
                                            i32 0,
                                            i32 0),
                 i32 8,
                 i8* getelementptr inbounds ([28 x i8]* @error_msg,
                                            i32 0,
                                            i32 0))
  unreachable

error_block3:           ; preds = %split_block
  call void (i8*, i8*, i32, i8*, ...)*
    @assert_fail(i8* getelementptr inbounds ([5 x i8]* @function,
                                            i32 0,
                                            i32 0),
                 i8* getelementptr inbounds ([12 x i8]* @filename,
                                            i32 0,
                                            i32 0),
                 i32 10,
                 i8* getelementptr inbounds ([28 x i8]* @error_msg,
                                            i32 0,
                                            i32 0))
  unreachable

declare void @assert_fail(i8*, i8*, i32, i8*)
```

This error block was created with two LLVM instructions: one call instruction to the assert-failure function and one terminating instruction, in this case the unreachable instruction, as illustrated in the code example above.

4.3 Measurements

A queuing port driver, running on both x86 and ARM 32-bit architectures, was used to conduct measurements on the different implementations.

The usage of a queuing port driver is to pass messages between a source and a destination point. The source can pass a message, of preferred size, which the queuing driver will put into a queue. The message can then be retrieved via the driver from this queue by the destination point. The performance of this driver is very memory intensive, since it will retrieve and write as a service, without impact from hardware, i.e. the overhead of performing checks prior to memory accesses is measured well.

The performed measurements were calling a queuing port driver from an application in user space to perform both a read and a write operation with different message sizes. To measure the performance decrease of the driver, exclusion of as much time as possible which was not effected by our implementation was performed.

The first measurement was done calling an empty function N times, denoted X. Secondly, we measured the time it took to invoke a system call N times with an empty function, denoted Y. Finally, the time for invoking the read and write operations to the queuing port driver N times were performed, denoted Z. This resulted in the time spent in driver, T_{driver}, being equal to:

$$T_{driver} = \frac{Z - Y - X}{N}$$

There were seven measurements performed on x86 and three on ARM, were the distinct measurements have been labeled with a character from 'a' to 'f'. An overview of the various measurements are presented in table 4.1 and table 4.2.

	1	2	3	4
a				
b	X		X	
c	X			X
d		X	X	
e		X		X

1 Check if address belongs to user space

2 Check if address belongs to kernel space

3 Provide an informative error message

4 Provide an uninformative error message

Table 4.1: Overview of the different features for the conducted measurements on the external function

	1	2
a		
e	X	
f		X

1 Perform check in external function

2 Inline instructions

Table 4.2: Overview of the two dynamic techniques used to prevent direct accesses to user space

For all these measurements we used the optimization flag -O2 which is a speed optimization that can be enabled in Clang. Measurements labeled 'b'-'f' did not check global or local variables. To see the impact of excluding the checks for local- and global variables, a measurement of implementation 'f' was performed where these checks were included. Also, the size was measured by retrieving the information from the section header of the generated ELF file [22].

4.4 Result

The results from the conducted measurements are presented below. For the used queuing port driver, we found 130 critical instructions, hence all LLVM passes inserted 130 checks. The first three figures present data performed on x86 and the last figure present data performed on ARM. In the following, we present data under the name *a* which represent the unaffected data for a single queuing port read- and write operation.

Figure 4.4, present the difference between the various features for the external method where an overview of them can be found in table 4.1. From this figure it is clear that implementation *e* is the fastest of arbitrarily different techniques. Therefore, we will only use this implementation in the following comparisons.

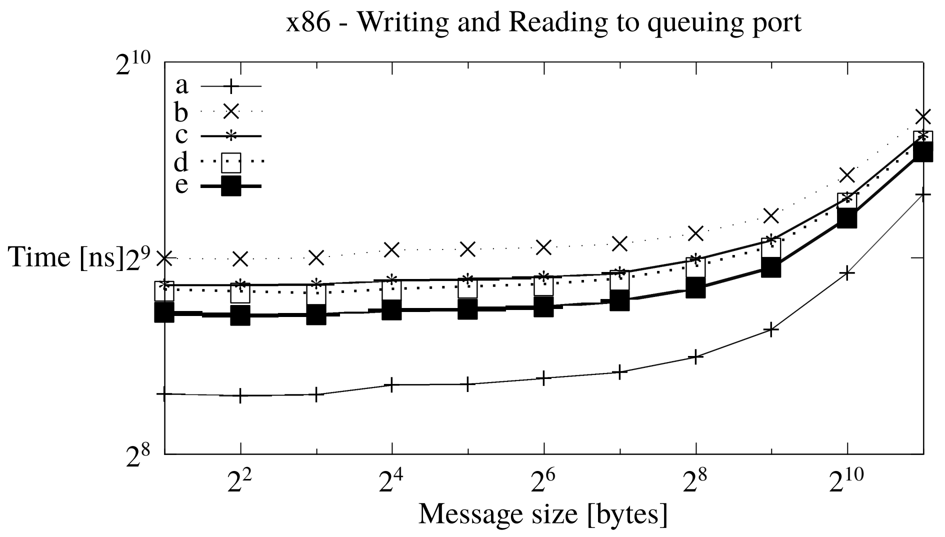


Figure 4.4: Difference between various techniques used for the external method

Figure 4.5 present the performance increase of the two different techniques, inlining instructions, *f*, and performing the check in an external function, *e*, where an overview of the different measurements can be found in table 4.2. The impact from the external function is quite high compared to the inline technique where the impact is extremely low. The low impact was expected since asm.js[13] showed that its feasible in their context.

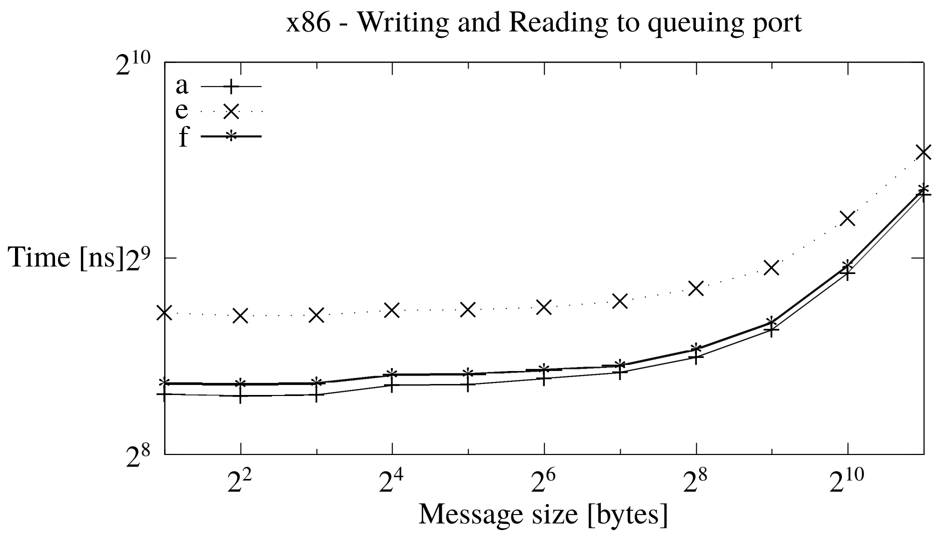


Figure 4.5: Performance increase of the two different techniques: inlining instructions, and performing the check in an external function on x86

The impact of excluding checks of local- and global variables is presented in figure 4.6. The difference is not noticeable in this driver, because it contains only few global and local variables.

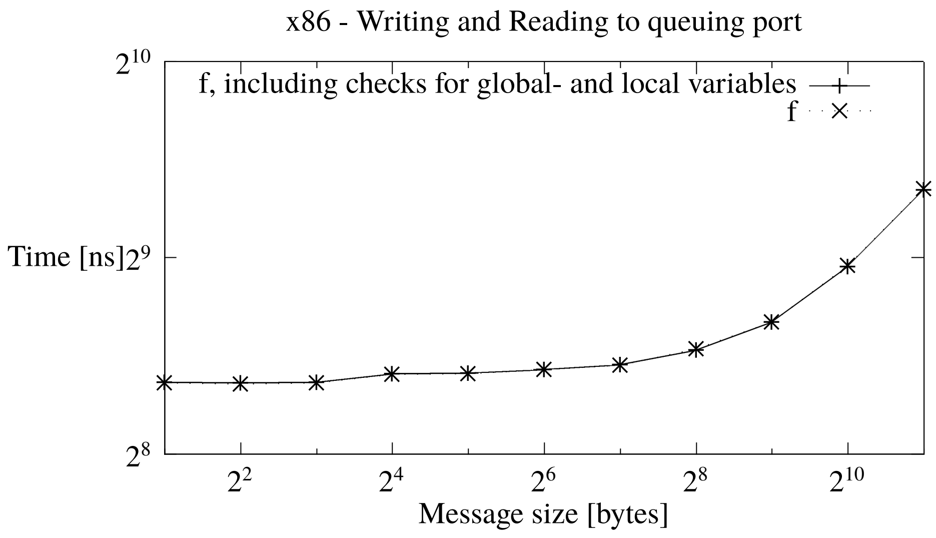


Figure 4.6: The impact of excluding checks of local- and global variables

The data collected on ARM is presented in figure 4.7, where we measured the performance increase of the two different techniques: inlining instructions and performing the check in an external function. The result is similar to the ones conducted on x86, i.e. inlining instructions has extremely low impact whereas the external function in quite noticeable.

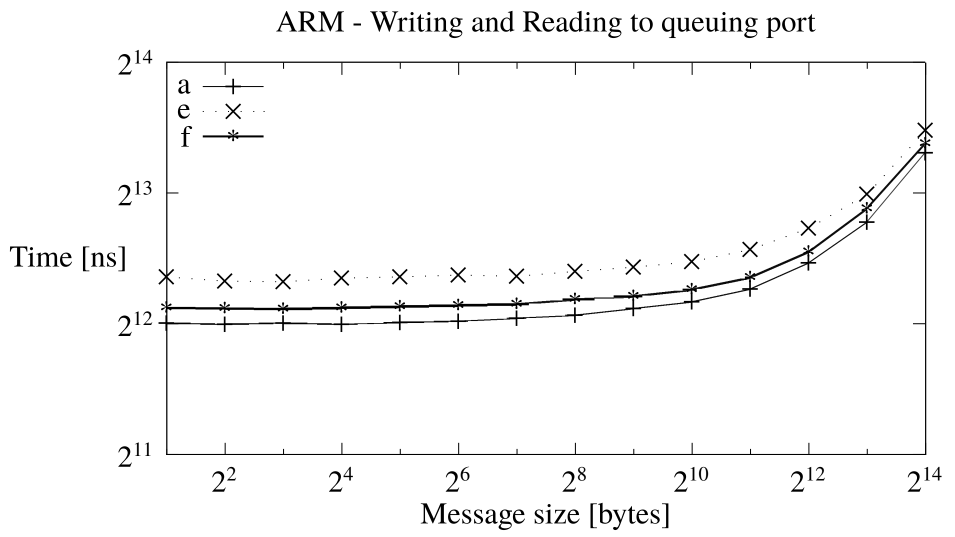


Figure 4.7: Performance increase of the two different techniques: inlining instructions and performing the check in an external function on ARM

Table 4.3 present the increase in time, measured as percentage, of a single queueing port read- and write operation with various message sizes (2 – 2048 bytes) on x86. The various techniques used for the implementation of the external methods, all appear to have a major impact on the performance, compared to the technique of inlining instructions. Inlinine instructions has a really low overhead, where the maximum impact is only 4.1%. The same illustration for the data conducted on ARM is presented in table 4.4, which also show a low overhead of only 9.2% for the inlining technique.

	Message size [bytes]										
	2	4	8	16	32	64	128	256	512	1024	2048
b	60.2	62.2	62.0	59.3	59.4	57.6	56.1	53.2	48.7	41.3	31.0
c	46.7	47.6	47.5	44.6	44.5	42.7	42.1	41.0	36.9	30.2	23.0
d	45.4	45.7	45.6	42.5	42.7	41.2	40.3	39.3	34.9	29.8	22.9
e	33.1	32.7	32.6	30.3	30.2	28.6	28.6	27.4	24.4	21.2	16.2
f	3.8	4.1	4.1	3.7	3.6	2.9	2.3	2.8	2.5	2.4	1.9

Table 4.3: Increase in time, measured as percentage of a single queueing port read- and write operation with various message sizes (2 – 2048 bytes) on x86

	Message size [bytes]										
	2	4	8	16	32	64	128	256	512	1024	2048
e	27.8	25.7	24.4	27.7	27.0	27.5	25.0	26.0	24.5	23.6	23.4
f	8.6	8.5	7.8	9.2	8.7	8.5	7.6	8.8	6.8	6.5	6.4

Table 4.4: Increase in time, measured as percentage of a single queueing port read- and write operation with various message sizes (2 – 2048 bytes) on ARM

Table 4.5 present the size of the executables (retrieved from the files section header), the number of inserted instructions as well as global variables in LLVM IR. The field *.text* describes the size of the executable instructions of the program and the global variables hold text characters for function, file and message which will be used in case of generating an error.

	.text [bytes]	# LLVM instructions	# global variables
a	6183	0	0
b	11833	260	13
c	7641	260	0
d	11833	260	13
e	7641	260	0
f	9363	780	13

Table 4.5: Size of the executables, number of inserted instructions and global variables in LLVM IR

See appendix A for the measured raw data.

Chapter 5

Static analyzes

This chapter will present sections related to the second concern, i.e. *Memory addresses must be checked for actually pointing to user space to prevent malevolent users to exploit the kernel driver to access kernel space for them.* The first section 5.1 presents the precautions from a theoretical aspect followed by section 5.2 which presents the implementations. Last, an illustrative code example will present how the different analyzes and extensions will countermeasure and work together to contribute to securing kernel drivers.

5.1 Accessing kernel space on behalf of user space

To prevent a malevolent user from exploiting a kernel driver to access kernel space for them, pointers must be used in a correct manner. In particular, pointers entered via system calls from user space must be checked before accessing their memory. A miss to do so may introduce serious security bugs into the system, as explained in section 2.2.5.4. After correctly performing checks on a pointer which is entered from user space, there are still some restrictions, recall the restriction explained in section 4.1. This work presents a solution which exploits the type system of Clang to attach information about pointers, in order to ensure correct behavior. The correctness of the behavior will, except from the pre-benefits from assigning different address spaces, conduct analyzes on the generated LLVM intermediate representation.

This approach can be divided into three main branches:

- introduction of new pointer types to differentiate pointers
- introduction of new functions which performs proper checks as well as cast between the introduced types

- LLVM analyzes and code transformations which verifies the correctness of the program

5.1.1 Introducing new pointer types

The need to introduce new types is twofold:

- distinguish checked- and unchecked user pointers
- distinguish user- and kernel pointers

This intuitively leads to three new types, where a pointer can be divided into a kernel- or a user pointer, and the user pointer can further be divided into a checked or an unchecked user pointer. This division can be illustrated with a tree construction, see figure 5.1.

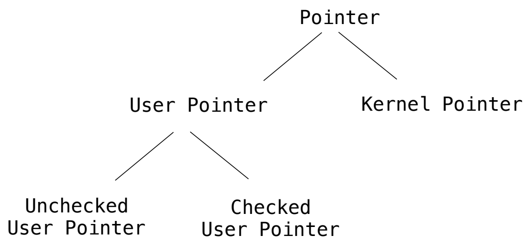


Figure 5.1: Division of different pointer types, illustrated in a tree structure

The problematic with this distinction is that drivers may be entered either from user space via system calls with user pointer or by another driver with kernel pointers. That means, that there need to be a common type such that both of these can enter.

In PikeOS, an invocation of a driver is always redirected through a kernel driver framework. That means that data entered from user space, will first enter the framework, see figure 5.2. All pointers which enters from user space, hence called *User Pointers*, to the framework must be distinguished from other pointers, since they are entered from an untrusted source.

When the pointers enter the kernel driver framework, they will be verified for their correctness. This means that the pointer will be verified so that it is pointing to its restricted area, i.e. to user space. Thereby, the pointers enter the framework as *Unchecked User Pointers* and after the verification they can be considered as *Checked User Pointers*. After that correctness is ensured, the invocation of the driver is performed, which is referred to as Kernel Driver B in figure 5.2.

On the other hand, if Kernel Driver B is requested for its service from another driver, in the same figure, figure 5.2, referred to as Kernel Driver A, the data flow is different. When a driver invokes another driver, it will as well go via the kernel driver framework. But, since these pointers will be passed from kernel space to kernel space, there is no need for the framework to check the pointers for their validity since they are entered from a trusted source. Thereby, the framework will only redirect the request to the correct driver.

That means, that Kernel Driver B can either be invoked from user space via a system call, where the pointers shall be considered as checked user pointers, or from another driver, where the pointers shall be considered as kernel pointers. This means that there need to be a new type which can be either a *Unsafe User Pointer* or *Kernel Pointer*, which we will call an *Unsafe Pointer*.

The pointer flow is even more complex. Since a pointer may enter drivers where the pointer is pointing to a data which in the driver will be interpret as an aggregated type, new pointers can enter as members, which is a very common situations for the system call ioctl.

If that is the case, when a user application invokes such call, then the kernel driver framework will check that the aggregated type itself relies in user space. The pointer to the aggregated type will then enter the driver. When that type is later copied in, from user- to kernel space, new data types may enter as members. If these members are of pointer type, it's critical to check their validity before its use. Since they have been copied in, they will now be pointers that relies in kernel space, but must point to user space. That means, they must be checked for pointing to user space, and after verification still be considered as user pointers.

Since the same function can be entered from another driver, the members may, in a correct way, point to kernel space. Thereby, pointers which are introduced after a copy operation can be a user- or a kernel pointer. This leads to a new type which ca be either an *Unchecked User Pointer* or a *Kernel Pointer* which we will refer to as an *Unchecked Pointer*

To conclude, the criticality of the pointer flow is especially two points. The first critical point is when data enters the kernel driver framework from user space via a system call, marked as *Security Boundary* in figure 5.2. User space shall never be trusted and thereby, all data must be handled carefully. It is thereby critical that the kernel driver framework performs proper checks on these pointers. The other critical point is when data is copied in, from user- to kernel space, since this operation may introduce new pointers as members. Compared with the first critical point, this is a bit more complex. At this point, it is not so clear how to perform proper checks, since the driver is unknown if it was invoked from user- or kernel space. Since it is a critical point if it's entered from user space, and not if it's entered from kernel space, we marked it as *Potential Security Boundary* in figure 5.2.

Because of the pointer flow, the initial pointer division, i.e. kernel pointer, unchecked user pointer and checked user pointer, is not possible. Instead, three new pointer types were introduced:

unchecked a pointer that is either pointing to user space or kernel space, that has not been checked for pointing to the expected place

unsafe a pointer that has been checked for either pointing to user space or kernel space

kernel a pointer which is created in kernel mode

Where the above rank the introduced types after how safe the pointers can be considered, starting with the least safe type. To introduce the three new types, we used the attribute address space where the different pointer types were assigned different address space. By using this approach, Clang will automatically distinguish the different types such that they are not mixed. Further, this attribute is visible also in the LLVM intermediate representation, which enables creation of analyzes based on them.

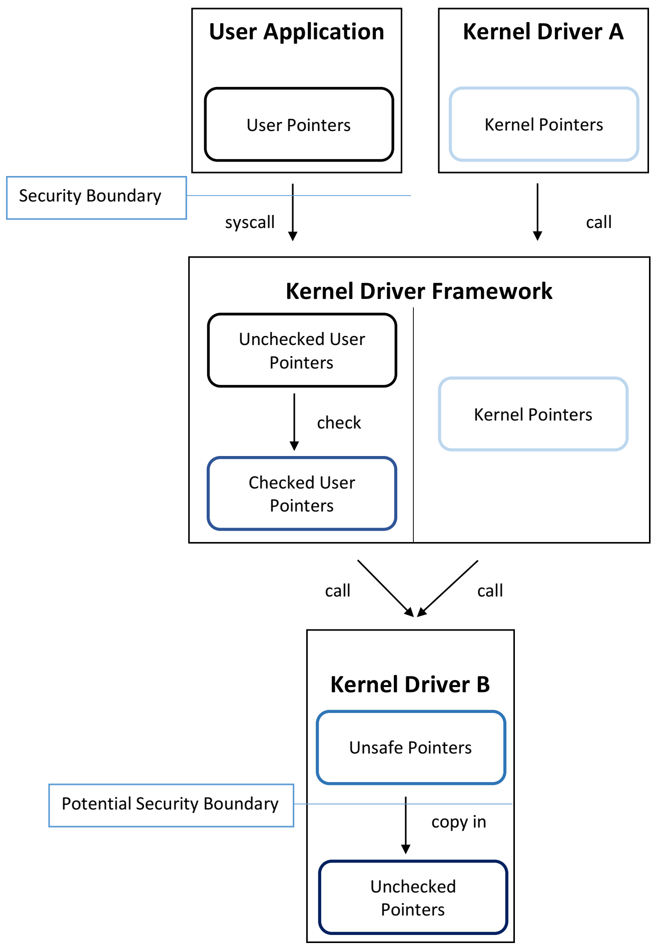


Figure 5.2: Pointer flow in PikeOS kernel drivers

5.1.2 Introducing new functions

The introduced pointer types can in some sense be seen as a state of a pointer. This means that a pointer assigned a specific type, it will most likely change its type throughout its life time.

Since the type of the pointer is essential for its permissions, the change of a type needs to be done in a controlled way. That means if a pointer is about to change its type to another type, then it needs to be checked that it is allowed to convert to the new type. There are four permitted type conversions:

1. unchecked → unsafe
2. kernel → unsafe
3. kernel → unchecked
4. unsafe → kernel

Type conversion number 1 is allowed for a pointer that is pointing to its expected memory area. An example for using this conversion is when a pointer is entered from user space via system call. That pointer would then have the type unchecked. For that pointer to change type, it needs to be checked for pointing to user space. If that is the case, then the pointer is allowed to be assigned from type unchecked to the new type unsafe.

A kernel pointer used in the same context a user pointer may be used, needs to convert to a lower safety type. This situation would apply to type conversions 2 and 3. If the same pointer later is used in a context where only kernel pointers are allowed, the type conversions number 1 and/or 4 would apply. For that to occur, a check needs to be performed.

For a pointer to be allowed to convert according to type conversion 4 the pointer will need to be checked to point to the memory area of the kernel. This may be a trivial check if the pointer earlier was typed kernel. But since the type unsafe is applicable for both pointers that come from user space and from kernel space, the check needs to be done.

The controlled way to convert between the different types can be implemented with four different functions.

The function for conversion 1 will need to be implemented in such way that if it's called with a pointer, provided from user space, it will check that it is pointing to user space. On the other hand, if it's called with a pointer that was provided from another kernel driver, the pointer is safe and no check is needed. If this test succeeds, then the pointer is allowed the new type, i.e. *unsafe*. To reduce instructions, the convert functions will be marked as an inline. The same goes for the other three type of convert functions.

Since the functions implementing conversion 2 and 3 will handle casts from a kernel type to a lower safety type, no check needs to be done. This will simply be a cast between two types. By making the function as an inline function, a call to this function will simply be represented as a cast between two address spaces in LLVM IR, which will later not result in any machine instruction.

The function implementing conversion 4, similarly to the function implementing conversion 1, will need to check the provided pointer before assigning it with the new type. The check needs to make sure that the pointer is pointing to kernel space.

5.1.2.1 Perform proper check

Since the function implementing the type conversion numbered 1 will need to know if the driver was entered from user space via a system call or from kernel space by another driver to be able to perform a proper check, it needs more consideration than the other three type convert functions.

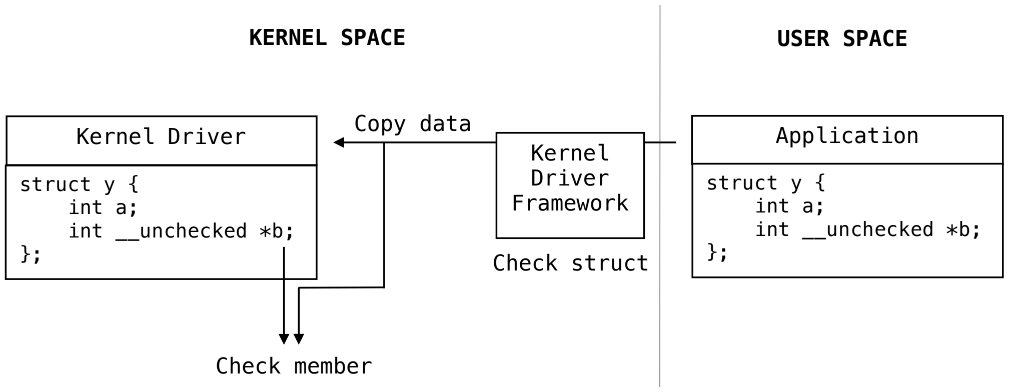
When an application needs a service from a driver, it will invoke a system call. This will then enter a function in kernel mode. This function can then control that pointers passed in from user space actually point to user space. In those situations, one will use the function implementing the conversion 1, from unchecked to unsafe. This check will is PikeOS be performed in the kernel driver framework, see figure 2.3.

After performing the check, the data may enter the actual driver to perform the requested service. The function entered in the driver will then take an *unsafe* pointer to some data in user space. Since the data is checked, it is safe to copy it from user into kernel space. If that data is interpreted as an aggregated type in the driver, checks on its members are needed.

The members shall be marked with the type *unchecked*, since these are not checked by the framework as the aggregated type. To perform a proper check on these, one should also use the function implementing the conversion 1.

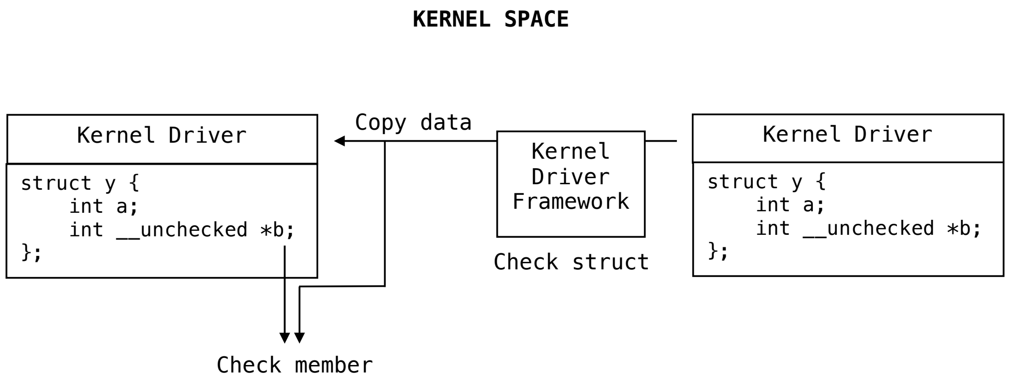
The function entered in the driver may as well be invoked by another driver. To invoke another driver which may also be entered via system calls, a need to change the type of the pointer to the data from kernel to unsafe is needed. When then entering the other driver, the same step as for the system call will be performed. Thereby, data will be copied in and checks will be performed if the data contains members of pointer type.

All pointers of type *unchecked* will be member pointers in the drivers. Since their aggregated type will either provided from user space or kernel space, one can use that information to perform a proper check. The aggregated type which points to user space shall only contain members which point to user space, and an aggregated type from kernel space is considered safe, hence no check is needed. With this information, proper checks can be performed. Figure 5.3 illustrates the case when data is provided via a system call from user space and figure 5.4 when a driver is invoked by another driver.



```
struct y {
    int a;
    int __unchecked *b;
};
```

Figure 5.3: Data provided via a system call



```
struct y {
    int a;
    int __unchecked *b;
};
```

Figure 5.4: Data provided from another device driver

5.1.3 LLVM Pass

By providing different address spaces to the differentiate pointers, analyzes can be enabled to extend the verification Clang enables. This is possible since the address space attribute is a visible type also in the LLVM intermediate representation. There are especially five things that is desired to check the correctness of:

- Check that the type of a pointer only is changed in the introduced functions. This is important to be able to ensure that proper checks are performed, see section 5.1.3.1.
- Check that members of pointer types is assigned with the right type, since these pointers will be introduced into kernel space after a copy operation, see section 5.1.3.2.
- Check that the right aggregated pointer is provided when checking a member of an aggregated type.
  - statically during compile time, see section 5.1.3.3.
  - insert checks for unchecked data automatically, see section 5.1.3.4.
- Check that there are no direct accesses to user space, see section 5.1.3.5.
- Check cast operations that introduces new pointers, see section 5.1.3.6.

5.1.3.1 Changing type through dedicated functions

The address space attribute assigned to a pointer will most likely change throughout its lifetime. But, when doing so, proper checks needs to be performed such that the new assigned type is indeed deserved. This is critical since the type of the pointer is the main thing deciding which operations that is allowed to be performed on this data.

It is therefore critical to verify, by analyzing the generated intermediate representation, that changes between the new types are only performed in the dedicated functions.

5.1.3.2 Members are marked unchecked

Pointers can be provided to a driver in three arbitrary ways. First, pointers can be provided as parameter to functions. Second, pointers can be introduced when cast operation are performed. Last, introduction can occur when pointers to aggregated types are provided as parameters to functions. If that type is later used in a copy operation, new pointer types may be introduced as members.

A system call, such as the ioctl, that takes a pointer to some data, will when it's declared assign the correct type of this pointer. Thereby, data provided via that function will be ensured to be assigned that type. This can be regulated in terms of an API.

If that pointer is interpret to point to a structure, it is critical to not forget to check its members. The members of aggregated types are neither checked by the framework nor ensured to be assigned the right, as the aggregated type that can be regulated by an API.

To ensure that all members are checked, an LLVM pass can be extended. This analyze would then check recursively that all pointer types within an aggregated type is assigned with the right type. By verifying that they are assigned with the correct type, it can also be stated that they will indeed be checked.

5.1.3.3 Analyze the parameters

There is one problem with the convert function that takes an *unchecked* pointer and verifies that it can be assigned the *unsafe* type. The problem is that it puts responsibility on the developer to provide the right aggregated type for the member they want to check. If the aggregated type is not the right one, but is from the same space as the actual aggregated type, there will be no problem. But if the developer provides an aggregated type that belongs to kernel space, whereas the actual aggregated type belongs to user space, things may end badly. It would therefore be helpful to analyze the intermediate representation to help the developer to use this function in the right way.

This could be done by creating a mapping of the member with its aggregated type when data is copied in. One problem with this approach is how to map these. Let's use the situation from figure 5.3 as an example. When the struct is copied in, a map of the member and the aggregated type should be created. In this case, the struct only contains one pointer type which should be mapped, i.e. the aggregated type of *int *b* is *struct y*. That means, later when using the dedicated convert function, see section 5.1.2, to check the pointer, one could check that it is providing *struct y* as its aggregated type.

Another situation could be that there are two different structs, structure A and structure B, both of type *struct y*. If one of those structures was passed in as a parameter, structure A, and the other was created in the function, structure B, then such simple mapping would fail. Structure A will then be copied in, and a mapping will be created. Later when using the two different types, the naive mapping will tell no difference between structure A and structure B, and therefore provide wrong warnings.

Another problem with such an easy mapping approach is if the developer uses a function call as the argument, which returns a pointer to the aggregated type. The return value from this call is then the necessary information which needs more complex non-function based analyzes. Those situations will also not work with the naive approach.

To summarize, providing such analyze is quite complex, and cannot be function based. A naive approach is not enough and further work is needed to provide accurate help in these situations for the developer.

5.1.3.4 Insert checks automatically

As described in section 5.1.3.3, pointers can be introduced after copy operations. To ensure that all these members are checked, a code transformation was created. This transformation automatically insert checks for all pointer members, that have not yet been checked, in the same point as the copy operation is performed. By doing so, it can be stated that all member pointers will be checked in a proper manner.

When data is copied in, all information that is needed to perform a check is available. I.e., the source of the copy as well as the pointers copied in are both available, so the new pointers can be verified to only point to user space if the source of the copy is user space. Thereby, one could directly check the data at that point. Such checks could then invalidate data that is violating their restrictions. By invalidating the data, it would then fail when an access later is done, and if no access is done to that data, then there will be no complications.

This approach should be easy to enable and disable for the kernel driver developer, since this will insert instructions into the code. If this is always enabled, then it may confuse the developer more than it would help. Confuse in the sense that the invalidation part is not in his power. It would not be a plain analysis of the code, it would in fact change it, but with the purpose to make it more secure. Also, this insertion would not be visible in the source code, since it’s inserted in the intermediate representation. Hence, it can be deceptive, and may even lead to increasing the time to find the actual problem of the code.

5.1.3.5 Direct memory access

The first part, section 4.1, of the thesis presents a way to check all addresses prior to accessing them. This prevents all direct accesses to user space. By using the introduced types, one could only allow direct accesses to pointers which are marked *kernel*. Since this would be based on the new types, the developer could get valuable information during compile time, which is of course very valuable since the approaches for the first part must run to find the critical code.

5.1.3.6 Performing cast operations

In C, casts between different types are frequently used. Some of these casts can introduce problems that will not be tackled by the type system explained above. One example of such situation is if an integer, which is assigned in user space, enters the kernel, where it’s later interpreted as a pointer. Since it’s not of pointer type when entering kernel space, it will not be checked.

The integer may then be used in the kernel driver in several ways. Some of these may introduce serious vulnerabilities:

```c
/* Provided via a system call form user space */
int value = 0x80000000;

/* Later use in kernel driver */
int *a = &value;          /* OK */
int *b = (int*)value;     /* NOT OK */
int **c = (int**)value;   /* NOT OK */
```

The first assignment will create a pointer *a* to the address of the value. Apart from not using the introduced types in a correct way, this is valid. The introduced types are not used in either of the assignments, and should be ignored.

The second assignment will interpret the integer as a pointer. The integer is assigned a value which if interpreted as an address, will point to an address. Thereby, the user can create a pointer to the desired address via an integer.

The third assignment will dereference the value twice, which in this example means that it will load the value at address 0x80000000. Thereby, the user can pass in integers to exploit or destroy kernel memory.

These are just a fraction of casts which may introduce serious security bugs. By using the type system in LLVM IR, casts can be found and prevented, where we will only focus on casts from integer to pointers.

5.2 Implementations

5.2.1 Introducing the LLVM-based fortification into PikeOS

The operating system PikeOS was used to implement the new types (by assigning them to different address spaces) as well as the described functions. These were introduced into the existing API.

All pointers passed in from user space to kernel space should be marked *unchecked*. When invoking a system call, the first function entered in the kernel belongs to the kernel driver framework. Those functions were extended in the framework which handles calls from user space, to only take *unchecked* pointers. The current checks were also changed such that the new convert function was used, see section 5.1.2. The check performed in the framework is not exhaustive, i.e. there are no checks for members if the pointer points to an aggregate type because the framework does not know the structure a driver uses for that aggregated type. Checks of those pointers must be done by the driver using our new technique.

Further, the API handling communication between the kernel drivers and the framework was extended with pointer types *unsafe* and *kernel*. This extension was not that trivial and also needed care since this is a base to the distinction between user- and kernel pointers in the driver.

Apart from the API, extension of the types to some existing service functions were performed, such as the functions performing the copy operations from/to user space.

5.2.2 LLVM Passes

The following LLVM Passes were implemented

- Changing type through dedicated functions
- Members are marked unchecked
- Insert a check automatically
- Direct memory access
- Integer to pointer

5.2.2.1 Changing type through dedicated functions

An LLVM pass which controlled that all conversions between address spaces were restricted to the four introduced functions was implemented.

Since the convert functions were implemented with a notation that makes a suggestion to the compiler to inline the instructions, a check for both situations where the instructions within the function was inlined and for situations were the instructions were not inlined, were needed.

The second situation will result in an address cast instruction which belongs to one of the functions. This enables an easy way to distinguish if it's performed in the dedicated functions or not. The implementation of the first situation was a little harder, since inlined instructions would result in an address space cast which could now belong to all functions.

The implementation of the first situation will therefore check if the instruction was inlined from the dedicated functions. This information was retrieved by enabling Clang to generate debug information, which resulted in a check that relies on that the debug information is enabled and that it will correctly generate information for all address space cast instructions.

Another approach would be to implement the convert functions without making the suggestion to the compiler to inline the instructions. This would imply only implementing the check for the second situation, i.e. no implementation which relies on the generated debug information. Although, the result of inlining the instructions may exclude instructions, which is the reason why we choose the implementation that relies on the debug information.

5.2.2.2 Members are marked unchecked

A pass was implemented to verify recursively that all members of type pointer are marked *unchecked* when an aggregated type is copied in from user- to kernel space.

The implementation includes the following aggregated types:

- Structure
- Array
- Pointer

To handle aggregated types which contain pointers to themselves, we verify that an aggregated type is only checked once. I.e., if the `struct` *list_t* in the code example presented below is used in a copy operation, the LLVM pass will recursively check its members. The aggregated type contains two members, both of type pointer and both correctly marked *unchecked.* Since the aggregated type contains an aggregated type itself, the LLVM pass will recursively check this as well to look for further pointer types. Since this member has already been checked, i.e. since it is also a `struct` *list_t*, the LLVM pass will not "unpack" it, preventing an infinite loop.

```c
typedef struct list_t list_t;

struct list_t {
    int __unchecked *a;
    list_t __unchecked *next;
};
```

One limitation with this implementation is that it does not include unions. As shown in section 2.3.2.4, unions are not supported in such way that they could be analyzed. Since a union is only presented in the intermediate representation with one of its members, there is no way to "unpack" such type to exploit all the members, because that information is simply not visible. This means that pointers in a union will not be checked to have the right type. This implies that if the developer is handling data of type union, one should pay extra attention to ensure that the data is used in a correct way and is not used in a context which could be exploited by a malevolent user to harm the system.

5.2.2.3 Insert a check automatically

The LLVM pass explained in section 5.2.2.2, that is verifying that all members are marked unchecked, was implemented with an optional extension.

That check unpacks all aggregate types and checks that members of pointer type are marked unchecked. The extension will also insert a check of that data at that point. This could be done since all data needed to perform the check is present, i.e. the aggregated variable and the data to be checked. The check uses, of course, the dedicated function to perform the check. If that function fails, it means that the member contains a pointer which does not point to the expected area, and shall therefore not be dereferenced. They shall not be dereferenced since they might have been assigned with a kernel address by a malevolent user. The result of dereferencing such pointer would be to access kernel space on behalf of user space, which must never be done. To prevent the developer from using the data, an assignment to an invalid address was inserted. On the other hand, if the convert function succeeds, the execution shall continue without any assignment.

5.2.2.4 Direct memory access

Prevention of direct memory accesses to user space was implemented by searching for memory access instructions. When finding such instruction, we checked that the address which will be accessed was related to a pointer assigned with the address space of a *kernel* type. If that was not the case, the user will be informed via a warning.

The limitation with this implementation is that it does not handle pointers to functions. In LLVM a pointer to a function is represented as a function type, and address spaces can only be assigned to pointer types.

5.2.2.5 Integer to pointer

A full analysis of whether a cast can be used by a malevolent user or not is not covered in this thesis. But, one check of such kind was introduced.

An LLVM pass which searches for casts from integer to pointer type was implemented. If a developer wants to perform such a cast, it should give the pointer the type *unchecked*. By providing it with the *unchecked* type, it will have to be checked prior to its use, which will result in preventing bugs.

5.2.3 Using the extension

An existing driver was adapted to the new types as well as the new functions. By doing this, we gained experience of how the proposed solution was to work with. All LLVM passes were enabled to help us extending the driver with the introduced types and functions. Apart from putting ourselves in the perspective of using the new features, this resulted in finding 4 bugs in the driver. Three of these were points where members within a structure were used before they were checked for their correctness to be a pointer to a location in user space.

Two of these were found in the error prone ioctl and one in the read function. The fourth bug was also found in the ioctl. This specific ioctl was implementing one command which should be used for requests from other drivers, where the bugs were a miss to check that the pointer was pointing to kernel space. Although this specific command was only supposed to be entered by another driver, it could as well have been entered via a system call from user space. Other from the filed bugs, we found that all generated warning messages were correct, i.e. verified by internal SYSGO developers, hence no false positives.

5.3 Code examples

Below illustrates different situations which can be found in an I/O control. The first example presents how the extensions are used correctly along with how the different static analyzes will verify the correctness, and for which row the different analyzes will focus on. Thereafter, code examples of limitations will be presented.

All example uses the introduced pointer types, which are provided with different address space, see section 5.1.1.

```c
#define __unchecked __attribute__((address_space(1)))
#define __unsafe __attribute__((address_space(2)))
#define __kernel __attribute__((address_space(0)))
```

All the examples presented that belongs to this section uses a configuration of 32-bit addresses, divided equally between user and kernel space. The addresses below 0x80000000 will herein belong to user space, and the addresses above, including the address itself, will belong to kernel space.

5.3.1 Correct usage

```c
struct s {
    int __unchecked *x_a;
};
```

```c
/* Simplified PikeOS code */
int my_ioctl(..., void __unsafe *u_data)
{
    /**
     * Since the default address space is 0,
     * below is equal to "int __kernel *i" */
    int *i;

    /* Cast void to struct, i.e. the right interpretation of the data */
    struct s __unsafe *u_my_struct = (struct s __unsafe *)u_data;

    /**
     * The struct has already been checked by the kernel driver
     * framework, and shall thereby not be checked again.
     * Since proper checks have been performed, it is safe to
     * copy the data from user to kernel space */
     struct s my_struct;
     if(drv_memcpy_in(&my_struct, u_my_struct, sizeof(my_struct))) {
```

```
19        /* Error: copy operation failed */
20     }
21
22     /**
23      * The members have not yet been checked,
24      * hence proper checks needs to be done before their use */
25     int __unsafe *u_a;
26     if(!drv_ptr_get_unsafe( &u_a,
27                             my_struct.x_a,
28                             sizeof(my_struct.x_a),
29                             u_my_struct)) {
30        /* Error: member is incorrect */
31     }
32
33     ...
34     /* print integer */
35     drv_put_d(*i);
36
37     ...
38 }
```

Row 2: API extension The created I/O control is following the extended API, where the pointer is provided with the *unsafe* type, since proper checks have been performed by the framework. See section 5.2.1.

Row 10: Changing type through dedicated functions Perform checks that there is no cast between the introduced types, which is not the case in this situation. The *unsafe* `void` pointer is correctly casted to an *unsafe* `struct` pointer. See section 5.1.3.1.

Row 18: Members are marked unchecked Check recursively that all members are marked correctly, i.e. are provided with the *unchecked* attribute. See section 5.1.3.2.

Row 18: Insert checks automatically Insert proper checks automatically for all pointer members. In case the function was entered from user space, check that the members are not provided with addresses that belongs to kernel space, and if so, assign them to invalid addresses. Thereby it will fail if they are later accessed, and if not there will be no complications. See section 5.1.3.4.

Row 26: Dedicated functions Using the dedicated functions correctly, which performs proper checks as well as convert to the new desired type. Verified with the analyze described in section 5.1.3.1.

Row 35: Direct memory access Accessing kernel space. Verified with static analyze described in section 5.1.3.5, i.e. that direct accesses are only performed on *kernel* pointers.

5.3.2 Limitation of providing the wrong aggregated variable

```
struct s {
    int __unchecked *x_a;
};
```

```
 1  /* Simplified PikeOS code */
 2  int my_ioctl(..., void __unsafe *u_data)
 3  {
 4      /**
 5       * Since the default address space is 0,
 6       * below is equal to int __kernel *i */
 7      int *i;
 8  
 9      /* Cast void to struct, i.e. the right interpretation of the data */
10      struct s __unsafe *u_my_struct = (struct s __unsafe *)u_data;
11  
12      /**
13       * The struct has already been checked by the kernel driver
14       * framework, and shall thereby not be checked again.
15       * Since proper checks have been performed, it is safe to
16       * copy the data from user to kernel space */
17       struct s my_struct;
18       if(drv_memcpy_in(&my_struct, u_my_struct, sizeof(my_struct))) {
19          /* Error: copy operation failed */
20       }
21  
22       /**
23        * The members have not yet been checked,
24        * hence proper checks needs to be done before their use */
25       int __unsafe *u_a;
26       if(!drv_ptr_get_unsafe( &u_a,
27                               my_struct.x_a,
28                               sizeof(my_struct.x_a),
29                               &my_struct)) {
30          /* Error: member is incorrect */
31       }
```

The source code example presented above illustrates a situation where the convert function which will check the member of the structure is provided with the wrong aggregated variable as its parameter. The right aggregated variable for this member is `u_my_struct`, as shown in section 5.3.1.

The aggregated variable provided in this case, is created in kernel space. Thereby, the check will trust the members which should never be done in case the function was invoked from user space via a system call.

By activating the LLVM pass which will automatically insert controls of all unchecked members directly when data is copied in, this will be prevented 5.1.3.4. That code transformation will directly at line 18 control if the member, in this case pointer *x_a*, is assigned to an address in user space if the function was entered from user space via a system call.

5.3.3 Limitations of cast operations

```c
struct s {
    int b;
};
```

```c
/* User space application */
struct s my_struct;
my_struct.b = 0x80000000;
ioctl(..., &my_struct);
```

```c
1  /* Simplified PikeOS code */
2  int my_ioctl(..., void __unsafe *u_data)
3  {
4      /**
5       * Since the default address space is 0,
6       * below is equal to int __kernel *i */
7      int *i;
8
9      /* Cast void to struct, i.e. the right interpretation of the data */
10     struct s __unsafe *u_my_struct = (struct s __unsafe *)u_data;
11
12     /**
13      * The struct has already been checked by the kernel driver
14      * framework, and shall thereby not be checked again.
15      * Since proper checks have been performed, it is safe to
16      * copy the data from user to kernel space */
17      struct s my_struct;
18      if(drv_memcpy_in(&my_struct, u_my_struct, sizeof(my_struct))) {
19         /* Error: copy operation failed */
20      }
21
22      int *p = (int *)my_struct.b;
23      int **q = (int **)&my_struct.b;
24 }
```

Cast operations are frequently used in C, where some of those might introduce major problems. At line 22, from the example presented above, a cast operation from an integer to a pointer is performed.

The integer member *b* of the structure was prior to the enter of this function assigned to the value 0x80000000 in user space by a malevolent user. Thereby, the assignment of the pointer *p* will imply an assignment to an address in kernel space, which was performed on behalf of a request invoked by a user. If this pointer is later dereferenced, the memory located at that specific address will be accessed. This situation can be prevented by activating the created LLVM pass presented in section 5.2.2.5.

Although, row 23 also present a cast operation, which are perfectly valid by the C standard but can imply complications. As presented in table 5.1, if the variable *q* is later dereferenced twice, memory in kernel space will be accessed. This is a limitation of the analyze and will not be prevented by this work.

Table 5.1 present an overview of the different operations which can be performed on the three created variables, used in the above example.

variable	&variable	variable	*variable	**variable
my_struct.b	0x0801EF34	0x80000000	**–**	**–**
p	0x805AFCE4	0x80000000	√	**–**
q	0x805AFCE0	0x0801EF34	0x80000000	↯

Table 5.1: Overview of the operations which can be performed on the three different variables created in the code example presented in this section. The table illustrates which operations are valid, which are prevented by the C type system, which are prevented with created analyzes and where there still are limitations

Description of symbols used in table 5.1

– Not allowed, prevented by the C type system

√ Prevented by the LLVM pass *Integer to pointer*

↯ Limitation of the static analyzes

5.3.4 Limitation of unions

```
struct s {
    int __unchecked *a;
    union {
        int b;
        int *c;
        float d;
    } u;
};
```

```
/* Simplified PikeOS code */
int my_ioctl(..., void __unsafe *u_data)
{
    /* Cast void to struct, i.e. the right interpretation of the data */
    struct s __unsafe *u_my_struct = (struct s __unsafe *)u_data;

    /**
     * The struct has already been checked by the kernel driver
     * framework, and shall thereby not be checked again.
     * Since proper checks have been performed, it is safe to
     * copy the data from user to kernel space */
     struct s my_struct;
     if(drv_memcpy_in(&my_struct, u_my_struct, sizeof(my_struct))) {
        /* Error: copy operation failed */
     }
    ...
}
```

Members of a structure have not been checked by the framework, and must therefor be checked in the respective driver. For the situation above, a structure enters which contains members of pointer type.

The LLVM pass described in section 5.1.3.2 will recursively verify that members within the structure only contains pointers that are correctly assigned with the attribute representing an *unchecked* type. The pass will correctly verify that the pointer *a* is assigned the right type, but no analyze will be performed on the union. Thereby, the developer will not be informed that the pointer member *c* is not assigned the correct type. Since only one member of a `union` is shown in the LLVM intermediate representation, there is no way to analyze such constructions, hence the result of the limitation of union usage. Although, it could be the case that it's the pointer member of the union that is the visible member in the intermediate representation, which in that case would result in informing the developer that the right type is not assigned.

5.3.5 Limitation of function pointer

```c
struct s {
    void (*f)(void);
};
```

```c
/* User space application */
void call_function();
 ...
int main()
{
    struct s my_struct;
    my_struct.f = call_function;
    ioctl(..., &my_struct);
    ...
}
```

```c
/* Simplified PikeOS code */
int my_ioctl(..., void __unsafe *u_data)
{
    /* Cast void to struct, i.e. the right interpretation of the data */
    struct s __unsafe *u_my_struct = (struct s __unsafe *)u_data;

    /**
     * The struct has already been checked by the kernel driver
     * framework, and shall thereby not be checked again.
     * Since proper checks have been performed, it is safe to
     * copy the data from user to kernel space */
     struct s my_struct;
     if(drv_memcpy_in(&my_struct, u_my_struct, sizeof(my_struct))) {
        /* Error: copy operation failed */
     }

    my_struct.f();
    ...
}
```

The example above presents code where a structure entered from user space to kernel space via a system call. The structure in question contains a member which is a pointer to a function. Before invoking the system call, this members was assigned to point to a function located in the application user space.

As presented on line 17, this function is later invoked from kernel space, which will imply a direct access from kernel space to user space. Since there is no possibility to assign the attribute address space to function pointers, this can not be prevented by the static analyze prevention, see section 5.1.3.2. To be able to catch such accesses, the dynamic approach needs do be enabled, see chapter 4.

Chapter 6

Discussions

This chapter is divided into two parts, where the first part presents the discussion related to the first concern and the second part relating the second concern.

6.1 Dynamic runtime checks

The first part shows how LLVM IR can be used to insert robustness assertions prior to accessing memory. This can be done in various ways, where two approaches were implemented. The measurements conducted on these implementations show that the practical implementation, all resulting in the same prevention, is important, see section 4.3 and 4.4.

From figure 4.5 conclusions can be drawn that inlining instructions to perform the check have significantly less impact on performance then performing it in an external file. This was expected, since it does exclude instructions and the check is only a few instructions. Table 4.3 shows that inlining of instructions only increases the execution time with a maximum of 4.1% while performing an external check increases the execution time with a maximum between 62.2 - 33.1%, depending on the C-implementation of the check.

The measured difference between the external functions is also noticeable. The difference between the two error messages can be explained by the overhead of passing parameters. The overhead of checking if the address belongs to user space instead of kernel space also shows that that there is a difference of how the check is performed. This can be due to how the compiler can optimize the check, cache-effects etc. The conclusion is that even though there are tiny differences in the C source code, it may have impact and should be considered if you want to achieve the least possible impact.

When looking at the difference from an implementation perspective, the external check is an easier approach. This only requires insertion of a call, with propitiated parameters, in the LLVM intermediate representation. The rest will then be performed in code written in C. The inlining approach needs more consideration when implementing, such as creating various blocks and causing right control flow between them. The implementation for the inlining approach was constructed to compare the address with a single value, to see if it belonged to the lower or upper part of the address space. This could be changed into an approach fitting both 32- and 64-bit architectures, dividing the addresses equally between the two spaces. By retrieving the highest bit of the address that is about to be accessed, that bit could be compared if it's equal to 0 or 1, which would provide the answer if it belongs to the upper or lower part. It is also important to think about the maintenance. LLVM is a project that is currently developed, and we are not even using the latest released version at the moment. Maintenance of inserting a call (and cast operation to get correct type of the parameters) does not seem to be much work. But, since the inlining approach uses more instructions, are splitting and creating basic blocks etc. it is more likely to need more maintenance. But, from looking at the fairly noticeable impact it had, we still prefer and recommend this approach over the external call.

It is also important to emphasize that the measurements are presenting the actual decrease of the execution time spent in the driver. That means that the impact when including the other factors, such as the time to perform a system call, is not that significant.

Figure 4.6 shows that for the queuing port driver, exclusion of performing checks on global- and local variables did not affect performance noticeably. The impact may be more significant on other drivers, and there is no reason to not exclude these checks.

The measured size and inserted code in LLVM IR of the kernel driver are presented in table 4.5. The comparison of the size and the increased time indicates that these are not always in relation to each other, i.e. a decreased code size does not always imply decreased execution time.

6.2 Static analyzes

This work shows that only a combination of the type system of Clang and the LLVM IR provides a safety net for kernel driver developers.

Some of the benefits with this approach is that it does not put pressure on the developer when adapting to the new types and functions, and that is is very easy to use. The procedure of adapting the types will be guided by just following the API as well as enabling the different analyzes. The same goes for the new functions. Also, the new functions shall not imply more code. Previously, there were check procedures similar to the ones introduced. The difference is that the new functions perform the check procedure as well as converting the pointers to correct introduced type. The cast between the introduced types will later not produce any machine instructions, i.e. there will be no additional code compared to the previous procedure.

The naive approach presented in section 5.1.3.3, is also interesting. It shows the limitations of function based analyzes. To compensate for this limitation, the invalidation approach were inserted 5.1.3.4. The compensation will handle misuse of the dedicated function, hence is a valuable safety net. However, we believe that it's more valuable to try to inform the developer during compile time. Those analyzes will directly inform where an error might occur, which means that you hopefully get more insight in where things might go wrong and also to prevent introducing such bugs later, thereby learning from your mistakes.

Performing casts in C is in many situations very helpful, but may also introduce serious vulnerabilities. The combination of allowing memory accesses through pointers makes it even more interesting. We believe that providing a completely watertight solution when using C is very complicated. One could always warn the developer when writing code which may later be used by a malevolent user, such as cast. But the problem is that many of these situations are frequently used. Therefore, warning for every possible C language vulnerability might result in to many warnings, which often result in losing the more important ones. Also, these kinds of problems are more related to a malevolent kernel driver developer than a malevolent user. To prevent the developer from themselves is a whole new area and is not treated in this thesis.

The limitations of LLVM 3.6 not supporting unions are of course noteworthy. This must of course be presented clearly to the developer using the tool.

Chapter 7
Conclusions and Future work

This chapter is divided into two sections. The first section will draw conclusions of the thesis and the last section presents our suggestions for future work.

7.1 Conclusions

This work fortifies kernel drivers with focus on two concerns.

The first concern is that there should never be a direct memory access from kernel- to user space. We propose a solution which prevent this by inserting robustness assertions prior to accessing memory in the LLVM intermediate representation. The inserted assertions were implemented in two ways, by inlining instructions and by invoking an external function which performed the check. The inlining of instructions was found to influence performance significantly smaller, in fact only with a maximum increase of 4.1%, and was implemented in a manner that generated valuable error messages. This shows that the dynamic runtime checks are indeed a feasible approach to be able to guarantee that there are no direct accesses to user space.

The second concern is preventing malevolent user from exploiting kernel drivers to access kernel space for them. We propose a solution which uses the type system of Clang combined with analyzes and code transformations on the generated LLVM IR code. The result was three new types and four introduced functions which we extended the PikeOS kernel driver API which we also implemented four static analyzes in form of LLVM passes, and one LLVM pass which inserted code to provide robustness. The LLVM passes were implemented as a tool which we extended SYSGOs IDE CODEO with. The provided tool is easy to use, has not shown any false positives, and uses a combination of techniques which provides a watertight solution in most cases. Most cases since it prevents malevolent users but is not a protection against malevolent driver developers. The result of this extension were that we were able to identify four bugs in a single driver.

7.2 Future work

To improve the second part of the thesis, one could explore later versions of LLVM. There may be an improvement concerning supporting unions. If so, this could be extended to the other passes. The current passes would then need to be explicitly tested so that they work correctly with the newer version.

There could also be more work to enable a pass which helps the developer provide the right aggregated variable, see section 5.1.3.3. This work would need deeper knowledge of control flows as well as more careful analyzes of how this could be written in C. We only presented a short introduction of how it can be complicated, for example in cases where the aggregated variable is returned from a function call.

This work is also restricted to pointers. It would be interesting to find approaches to extend the analyzes to more types, such as function pointers. We are limited in our approach since we are using the address space type, which only applies to data pointers. It would be interesting to find a way to trace also function pointers, and looking further into how other types may cause vulnerabilities in the kernel.

Further tests would also need to be conducted. Currently, we have concentrated on the x86 and ARM 32-bit architectures. This could be extended to support especially 64-bit architectures but also for others than x86 and ARM, such as the Power PC.

Bibliography

[1] D. P. Bovet and M. Cesati. *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[2] S. Bugrara and A. Aiken. "Verifying the safety of user pointer dereferences". In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE. 2008, pp. 325–338.

[3] A. Chou, B. Fulton, and S. Hallem. *Linux Kernel. Security Report.* `http://www.coverity.com/library/pdf/coverity_linuxsecurity.pdf`. [Online; accessed 6-May-2016]. 2005.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. *An empirical study of operating systems errors*. Vol. 35. 5. ACM, 2001.

[5] T. Coudray, A. Fontaine, and P. Chifflier. "Picon: Control Flow Integrity on LLVM IR". In: (2015).

[6] S. P. Dandamudi. *Guide to RISC processors: for programmers and engineers*. Springer Science & Business Media, 2005.

[7] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. "Memory safety without garbage collection for embedded applications". In: *ACM Transactions on Embedded Computing Systems (TECS)* 4.1 (2005), pp. 73–111.

[8] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. "Memory safety without runtime checks or garbage collection". In: *ACM SIGPLAN Notices* 38.7 (2003), pp. 69–80.

[9] S. Engle and S. Whalen. "Finding User/Kernel Pointer Bugs in FreeBSD". In: (2005).

[10] A. Forin, D. Golub, and B. N. Bershad. *An I/O system for Mach 3.0.* Carnegie-Mellon University. Department of Computer Science, 1991.

[11] J. Foster. *CQUAL: A tool for adding type qualifiers to C.* `http://www.cs.umd.edu/~jfoster/cqual/.`. [Online; accessed 6-May-2016].

[12] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. *The performance of μ-kernel-based systems*. Vol. 31. 5. ACM, 1997.

[13] D. Herman, L. Wagner, and A. Zakai. *asm.js*. `http://asmjs.org/spec/latest/`. [Online; accessed 6-May-2016].

[14] R. Johnson and D. Wagner. "Finding User/Kernel Pointer Bugs with Type Inference." In: *USENIX Security Symposium*. Vol. 2. 2004,

[15] T. Jones. *User space memory access from the Linux kernel*. `http://www.ibm.com/developerworks/library/l-kernel-memory-access/`. [Online; accessed 6-May-2016]. 2010.

[16] B. W. Kernighan, D. M. Ritchie, and P. Ejeklint. *The C programming language*. Vol. 2. prentice-Hall Englewood Cliffs, 1988.

[17] C. Lattner and V. Adve. "Architecture for a next-generation gcc". In: *GCC Developers Summit*. Citeseer. 2003, p. 121.

[18] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[19] C. Lattner and V. Adve. *LLVM Language Reference Manual*. `http://llvm.cs.uiuc.edu/docs/LangRef.html.`. [Online; accessed 6-May-2016].

[20] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. "WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code". In: *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE. 2009, pp. 43–52.

[21] B. C. Lopes and R. Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing Ltd, 2014.

[22] W. Nishida. *Executable and Linkable Format (ELF)*. `http://www.skyfree.org/linux/references/ELF_Format.pdf`. [Online; accessed 7-May-2016].

[23] S. Peiró, M. Muñoz, M. Masmano, and A. Crespo. "Detecting stack based kernel information leaks". In: *International Joint Conference SOCO'14-CISIS'14-ICEUTE'14*. Springer. 2014, pp. 321–331.

[24] *Pike OS*. `https://en.wikipedia.org/wiki/PikeOS`. [Online; accessed 7-May-2016].

[25] A. Rubini and J. Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[26] O. O. Ruwase. "Improving Device Driver Reliability through Decoupled Dynamic Binary Analyses". In: (2013).

[27] J. Skeppstedt and C. Söderberg. *Writing efficient C code: a thorough introduction for Java programmers: covers C99, the Standard C library and the new CIX draft*. Skeppberg, 2011.

[28] SYSGO. *PikeOS User Manual (Fundamentals)*.

[29] SYSGO. *Products: PikeOS Hypervisor*. `https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/`. [Online; accessed 7-May-2016].

[30] T. C. Team. *Clang 3.6 documentation: Clang Compiler User's Manual*. `http://llvm.org/releases/3.6.2/tools/docs/UsersManual.html`. [Online; accessed 6-May-2016].

[31] L. Torvalds. *Sparse: A semantic parser for C*. `http://sparse.wiki.kernel.org.`. [Online; accessed 7-May-2016]. 2006.

[32] J. Yang, T. Kremenek, Y. Xie, and D. Engler. "MECA: an extensible, expressive system and language for statically checking security properties". In: *Proceedings of the 10th ACM conference on Computer and communications security*. ACM. 2003, pp. 321–334.

[33] A. Zakai. "Emscripten: an LLVM-to-JavaScript compiler". In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2011, pp. 301–312.

Appendices

Appendix A
Data from measurements

This appendix holds the raw measured data of a single queuing port read and write operation. The tables and figures in section 4.4 are constructed from this data.

Size [bytes]	a [ns]	b [ns]	c [ns]	d [ns]	e [ns]	f [ns]
2	317	508	465	461	422	329
4	315	511	465	459	418	328
8	316	512	466	460	419	329
16	327	521	473	466	426	339
32	328	523	473	468	427	340
64	335	528	478	473	431	345
128	342	534	486	480	440	350
256	361	553	509	503	460	371
512	398	592	545	537	495	408
1024	486	687	633	631	589	498
2048	641	840	789	788	745	653
4096	951	1150	1101	1098	1057	966
8192	1582	1785	1732	1731	1688	1597
16384	3258	3437	3386	3371	3341	3265
32768	6444	6630	6583	6573	6539	6458
65536	12618	12812	12770	12748	12705	12631
131072	26334	26566	26469	26486	26432	26329
262144	52639	52866	52807	52825	52765	52684
524288	104870	105110	105038	105058	105002	104908

Table A.1: Measured time, in nanoseconds [ns], of a single queuing port read- and write operation on x86 with different message sizes, in bytes.

Size [bytes]	**a [ns]**	**e [ns]**	**f [ns]**
2	4103	5245	4455
4	4088	5139	4435
8	4111	5113	4431
16	4082	5214	4456
32	4128	5242	4488
64	4152	5293	4507
128	4218	5273	4539
256	4286	5402	4664
512	4440	5529	4742
1024	4605	5691	4904
2048	4922	6072	5238
4096	5655	6787	5998
8192	7026	8143	7522
16384	10150	11432	10629
32768	24313	25448	24710
65536	44623	45775	45008
131072	85004	86199	85338
262144	165785	166953	165987
524288	327448	328459	327173

Table A.2: Measured time, in nanoseconds [ns], of a single queuing port read- and write operation on ARM with different message sizes, in bytes.

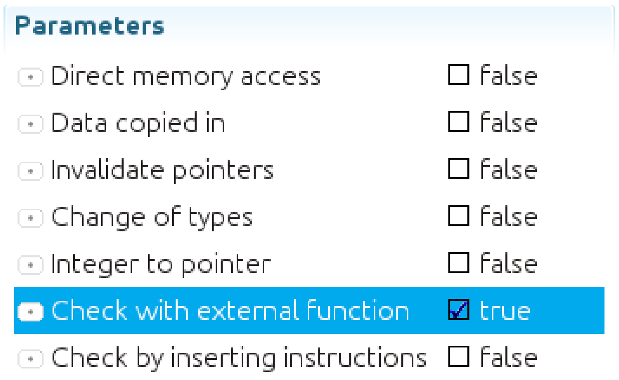
Appendix B
CODEO extensions

The different implementations presented in this work is integrated into SYSGOs Eclipse based IDE, CODEO. When creating a new kernel driver, the various analyzes and code transformations can easily be enabled and disabled in the configuration file that is automatically provided.

B.1 Call external function

The code transformation presented in 4.2.1 can be enabled by checking the 'Check with external function', which is present in the created configuration file for kernel drivers, see figure B.1. To illustrate how this code transformation works for developers, the code example in figure B.2 will be used. In this example, a structure enters from user- to kernel space which has an integer pointer as its member. If this pointer has been assigned, in user space, to point to some integer that relies in the application, the dereferencing will imply a direct access to user space. This driver will dereference it by printing the integer that the member is pointing to, which is performed by the PikeOS function *drv_put_d*.

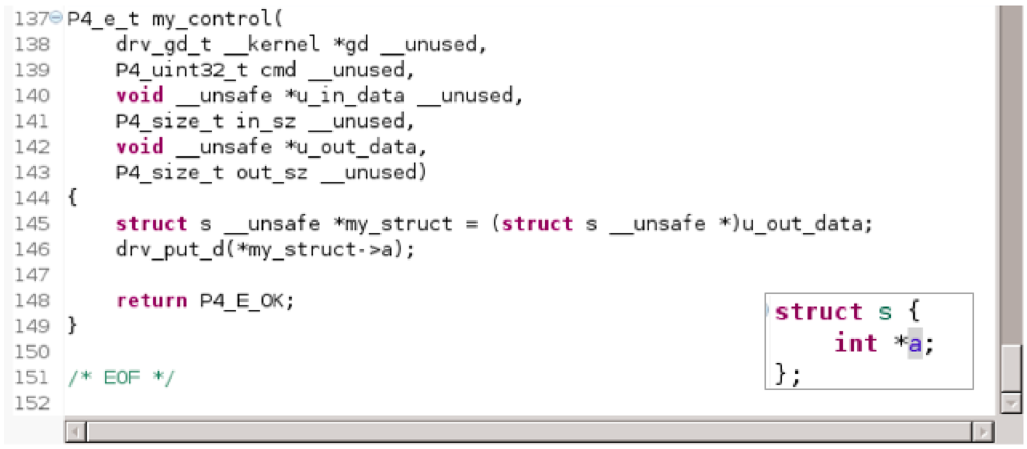
By enabling the created LLVM pass, this access will fail during runtime, causing the presented error message in figure B.3. As seen from this message, it will present that the error occurred in *check_function.c* on line 20, which is the location to where the check is performed, i.e. not the actual memory access that is on line 146.



Parameters

Direct memory access	☐ false
Data copied in	☐ false
Invalidate pointers	☐ false
Change of types	☐ false
Integer to pointer	☐ false
Check with external function	☑ true
Check by inserting instructions	☐ false

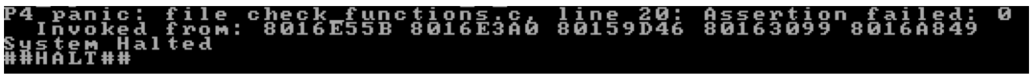
Figure B.1: Enable LLVM pass which fully prevents direct accesses to user space by invoking an external function



```
137 P4_e_t my_control(
138     drv_gd_t __kernel *gd __unused,
139     P4_uint32_t cmd __unused,
140     void __unsafe *u_in_data __unused,
141     P4_size_t in_sz __unused,
142     void __unsafe *u_out_data,
143     P4_size_t out_sz __unused)
144 {
145     struct s __unsafe *my_struct = (struct s __unsafe *)u_out_data;
146     drv_put_d(*my_struct->a);
147
148     return P4_E_OK;
149 }
150
151 /* EOF */
152
```

```
struct s {
    int *a;
};
```

Figure B.2: Example code which performs a direct access to user space

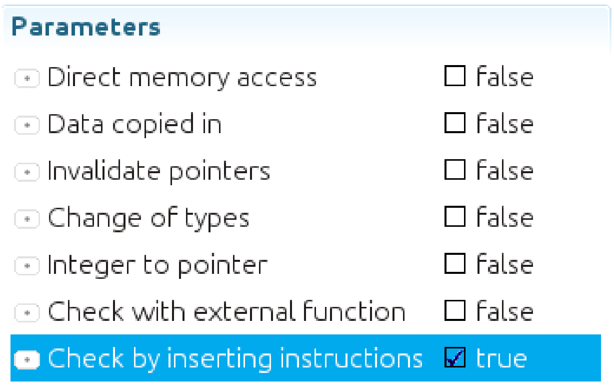


```
P4 panic: file check_functions.c, line 20: Assertion failed: 0
  Invoked from: 8016E55B 8016E3A0 80159D46 80163099 8016A849
System Halted
##HALT##
```

Figure B.3: Generated runtime error that prevent direct access to user space, that is prevented by invoking an external function

B.2 Inline instructions

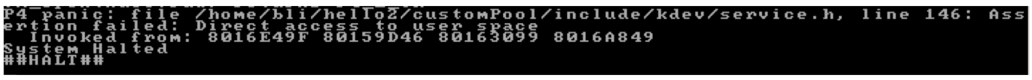
To prevent direct accesses to user space by inlining LLVM instructions, recall section 4.2.2, 'Check by inserting instructions' can be enabled, see figure B.4. When using the same code example as the one presented in section B.1, then the error message presented in figure B.5 would occur. As can bee seen from this figure, the error leads to the correct line, i.e. to line 146, which is where the actual access occurs. Although, the file is not correct, which is the intention of this approach. This leads to the file where the PikeOS function *drv_put_d* is implemented and not to the driver itself.



Parameters

Direct memory access	☐ false
Data copied in	☐ false
Invalidate pointers	☐ false
Change of types	☐ false
Integer to pointer	☐ false
Check with external function	☐ false
Check by inserting instructions	☑ true

Figure B.4: Enable LLVM pass which fully prevents direct accesses to user space by inlining LLVM instructions

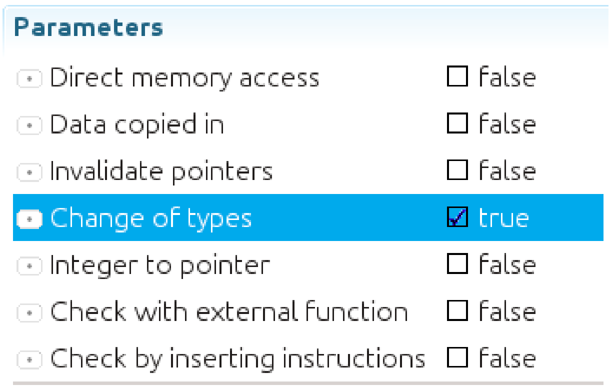


```
P4 panic: file /home/bli/hello2/customPool/include/kdev/service.h, line 146: Ass
ertion failed: Direct access to user space
  Invoked from:  8016E49F 80159D46 80163099 8016A849
System Halted
##HALT##
```

Figure B.5: Generated runtime error that prevent direct access to user space, that is prevented by inlining LLVM instructions

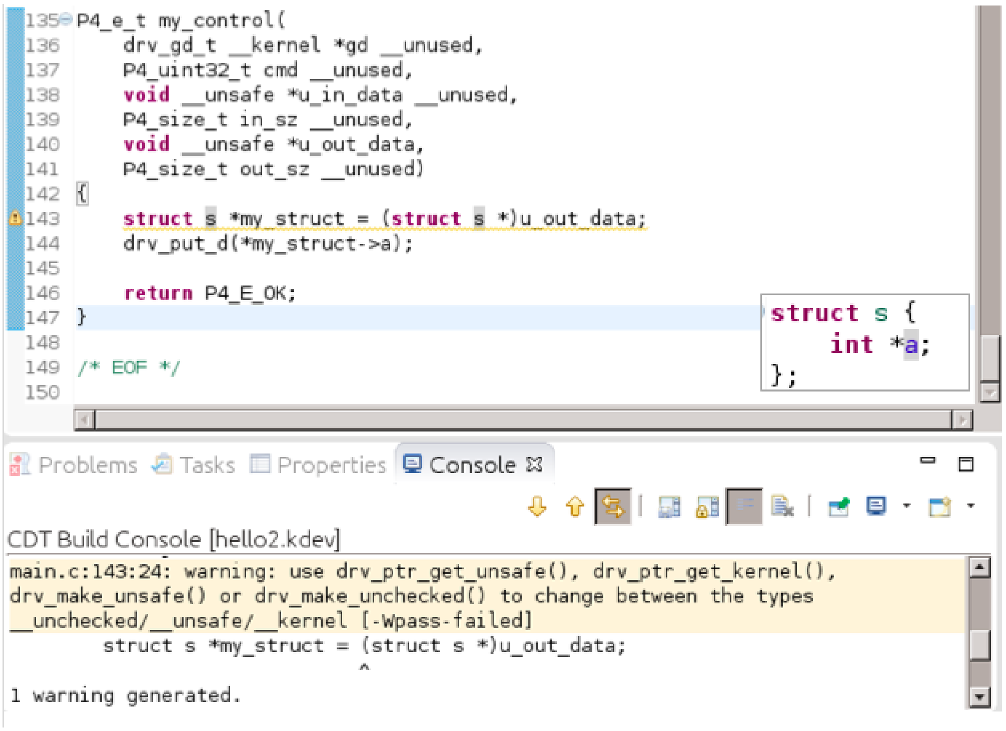
B.3 Changing type through dedicated functions

Changes between the introduced types should only be done through the dedicated functions which also performs proper checks, recall section 5.2.2.1. This can be prevented by enabling the option ’Change of types’ in CODEO, see figure B.6. An example of a situation where this may occur is presented in figure B.7. The driver takes a pointer to some data, which is here interpreted as a structure. To interpret the entered data as a pointer to a structure instead of a void pointer, a cast is performed. Although, this cast operation implies that the new pointer type, i.e. *unsafe*, that is provided to the entered void data would be changed in to the *kernel* type, since this type is provided with the default address space. Therefore, this will change from a potential user space pointer to a kernel pointer, which may imply serious problems. For example, if it’s later accessed directly. By enabling the created LLVM pass, this would during compile time inform the developer via a warning that the dedicated functions should be used, as can be seen in figure B.7.



Parameters	
Direct memory access	☐ false
Data copied in	☐ false
Invalidate pointers	☐ false
Change of types	☑ true
Integer to pointer	☐ false
Check with external function	☐ false
Check by inserting instructions	☐ false

Figure B.6: Enable LLVM pass which checks that changes between the introduced types are done through the introduced functions



```
P4_e_t my_control(
    drv_gd_t __kernel *gd __unused,
    P4_uint32_t cmd __unused,
    void __unsafe *u_in_data __unused,
    P4_size_t in_sz __unused,
    void __unsafe *u_out_data,
    P4_size_t out_sz __unused)
{
    struct s *my_struct = (struct s *)u_out_data;
    drv_put_d(*my_struct->a);

    return P4_E_OK;
}

/* EOF */
```

```
struct s {
    int *a;
};
```

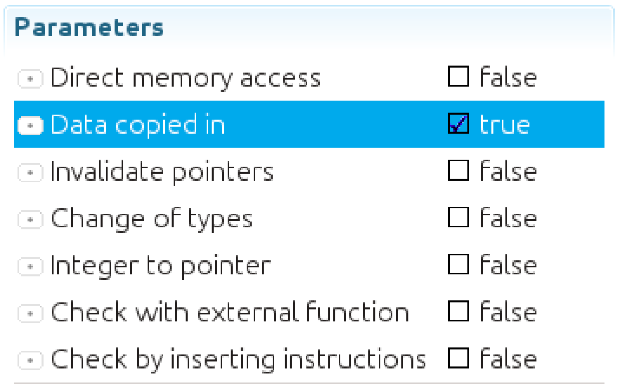
CDT Build Console [hello2.kdev]

```
main.c:143:24: warning: use drv_ptr_get_unsafe(), drv_ptr_get_kernel(),
drv_make_unsafe() or drv_make_unchecked() to change between the types
__unchecked/__unsafe/__kernel [-Wpass-failed]
        struct s *my_struct = (struct s *)u_out_data;
                              ^

1 warning generated.
```

Figure B.7: Example code which changes the introduced types by performing simple cast operations, i.e. not through the introduced functions

B.4 Members are marked unchecked

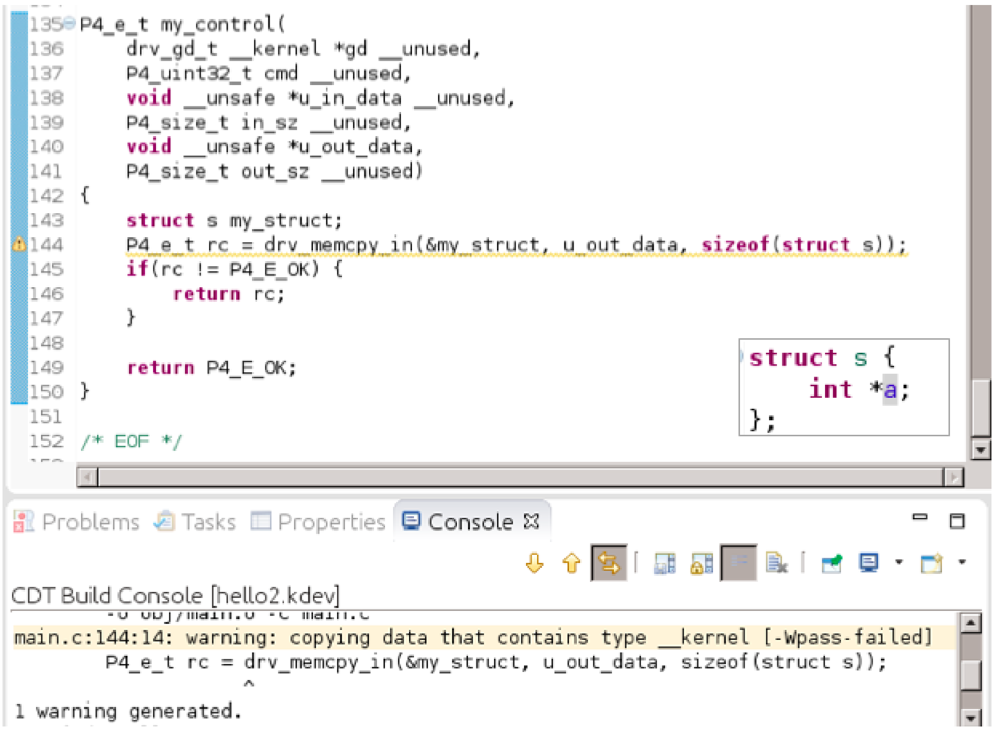
New pointers can be introduced in the kernel drivers when a copy operation is performed. These pointers have not been checked, and should therefore be provided with the *unchecked* attribute, as explained in section 5.2.2.2. To enable a static analyze that vefifies that this is indeed correct, the option 'Data copied in' can be enabled, see figure B.8. Consider the example presented in figure B.9 where a structure is copied in from user- to kernel space with the PikeOS function *drv_memcpy_in*. This structure has been checked by the kernel driver framework and may therefore correctly be copied in. The member of this structure is a pointer, which is provided with the *kernel* type, since that type is provided with the default address space. If this function was invoked via a system call, then it's crucial to check that all members also point to user space, otherwise the user may use the driver to access kernel space for them. As can be seen in this figure, a warning is generated to inform the developer that data is copied in which is marked with the wrong type, i.e. in this case *kernel*.



Parameters

Direct memory access	☐ false
Data copied in	☑ true
Invalidate pointers	☐ false
Change of types	☐ false
Integer to pointer	☐ false
Check with external function	☐ false
Check by inserting instructions	☐ false

Figure B.8: Enable LLVM pass which checks that all members are marked unchecked when data is copied in



```
P4_e_t my_control(
    drv_gd_t __kernel *gd __unused,
    P4_uint32_t cmd __unused,
    void __unsafe *u_in_data __unused,
    P4_size_t in_sz __unused,
    void __unsafe *u_out_data,
    P4_size_t out_sz __unused)
{
    struct s my_struct;
    P4_e_t rc = drv_memcpy_in(&my_struct, u_out_data, sizeof(struct s));
    if(rc != P4_E_OK) {
        return rc;
    }

    return P4_E_OK;
}

/* EOF */
```

```
struct s {
    int *a;
};
```

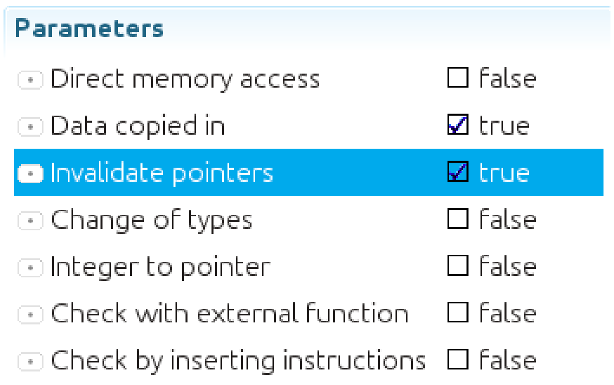
CDT Build Console [hello2.kdev]

```
         -o obj/main.o -c main.c
main.c:144:14: warning: copying data that contains type __kernel [-Wpass-failed]
        P4_e_t rc = drv_memcpy_in(&my_struct, u_out_data, sizeof(struct s));
                    ^
1 warning generated.
```

Figure B.9: Example code which copies in data that contains members that are not marked unchecked

B.4.1 Insert checks automatically

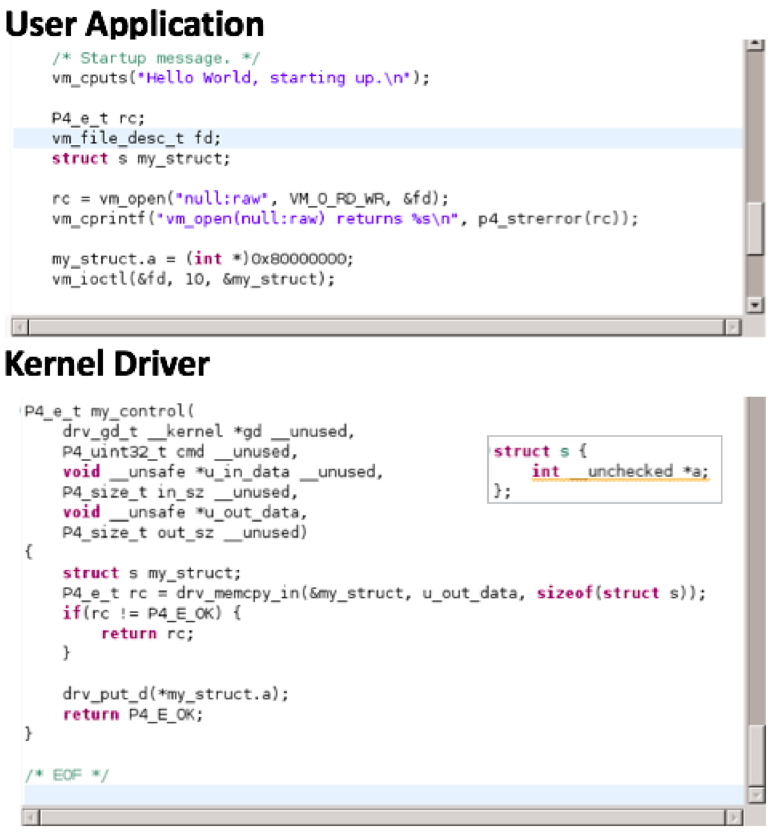
The option 'Data copied in' comes with an additional option 'Invalidate pointer' which will directly check all pointer members that are unchecked when data is copied in, see figure B.10. To illustrate an example of how this will prevent users from accessing kernel space, the code example presented in figure B.11 will be used. The example present two views, the top view present a user application where a structure is created. The structure contains a pointer to an integer, as can be seen in the right corner of the bottom view. The application assigns this pointer to the address *0x80000000* which in the current configuration is an address that belongs to kernel space. This pointer then enter as a member to the ioctl. The driver invoked is presented in the bottom view of the same figure. Since the structure is checked by the framework, it can correctly be copied in from user- to kernel space by the PikeOS function *drv_memcpy_in*. Since the 'Invalidate pointer' is enabled, the created LLVM pass will insert checks for all unchecked member pointers when data is copied in. In case the check fails, then the pointer will be assigned to an invalid address, see section 5.2.2.3, which is the case for the presented example. Therefore, when that pointer is later dereferenced, in the *drv_put_d* function, the access will fail, resulting in the runtime error presented in figure B.12.



Parameters

Direct memory access	☐ false
Data copied in	☑ true
Invalidate pointers	☑ true
Change of types	☐ false
Integer to pointer	☐ false
Check with external function	☐ false
Check by inserting instructions	☐ false

Figure B.10: Enable LLVM pass which automatically perform proper checks on all members that are marked unchecked when data is copied in

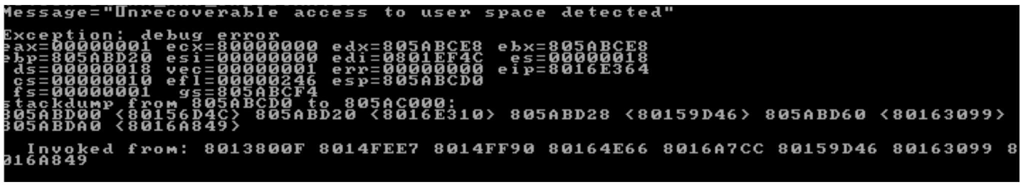


```
/* Startup message. */
vm_cputs("Hello World, starting up.\n");

P4_e_t rc;
vm_file_desc_t fd;
struct s my_struct;

rc = vm_open("null:raw", VM_O_RD_WR, &fd);
vm_cprintf("vm_open(null:raw) returns %s\n", p4_strerror(rc));

my_struct.a = (int *)0x80000000;
vm_ioctl(&fd, 10, &my_struct);
```

```
P4_e_t my_control(
    drv_gd_t __kernel *gd __unused,
    P4_uint32_t cmd __unused,
    void __unsafe *u_in_data __unused,
    P4_size_t in_sz __unused,
    void __unsafe *u_out_data,
    P4_size_t out_sz __unused)
{
    struct s my_struct;
    P4_e_t rc = drv_memcpy_in(&my_struct, u_out_data, sizeof(struct s));
    if(rc != P4_E_OK) {
        return rc;
    }

    drv_put_d(*my_struct.a);
    return P4_E_OK;
}

/* EOF */
```

```
struct s {
    int __unchecked *a;
};
```

Figure B.11: Example code which access members which have not yet been checked

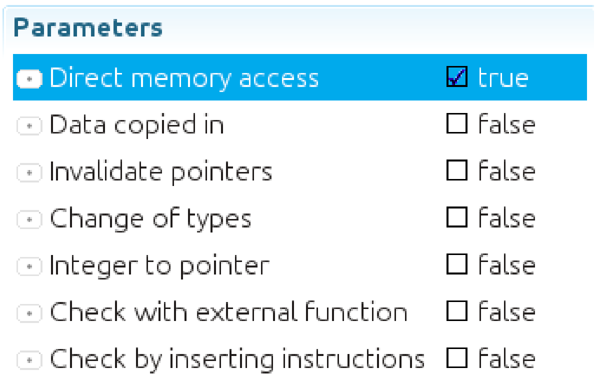


```
Message="Unrecoverable access to user space detected"

Exception: debug error
eax=00000001 ecx=80000000 edx=805ABCE8 ebx=805ABCE8
ebp=805ABD20 esi=00000000 edi=0801EF4C  es=00000018
 ds=00000018 vec=00000001 err=00000000 eip=8016E364
 cs=00000010 efl=00000246 esp=805ABCD0
 fs=00000001  gs=805ABCF4
stackdump from 805ABCD0 to 805AC000:
805ABD00 <80156D4C> 805ABD20 <8016E310> 805ABD28 <80159D46> 805ABD60 <80163099>
805ABDA0 <8016A849>

  Invoked from: 8013800F 8014FEE7 8014FF90 80164E66 8016A7CC 80159D46 80163099 8
016A849
```

Figure B.12: Generated runtime error that prevent accessing data which is incorrect

B.5 Direct memory access

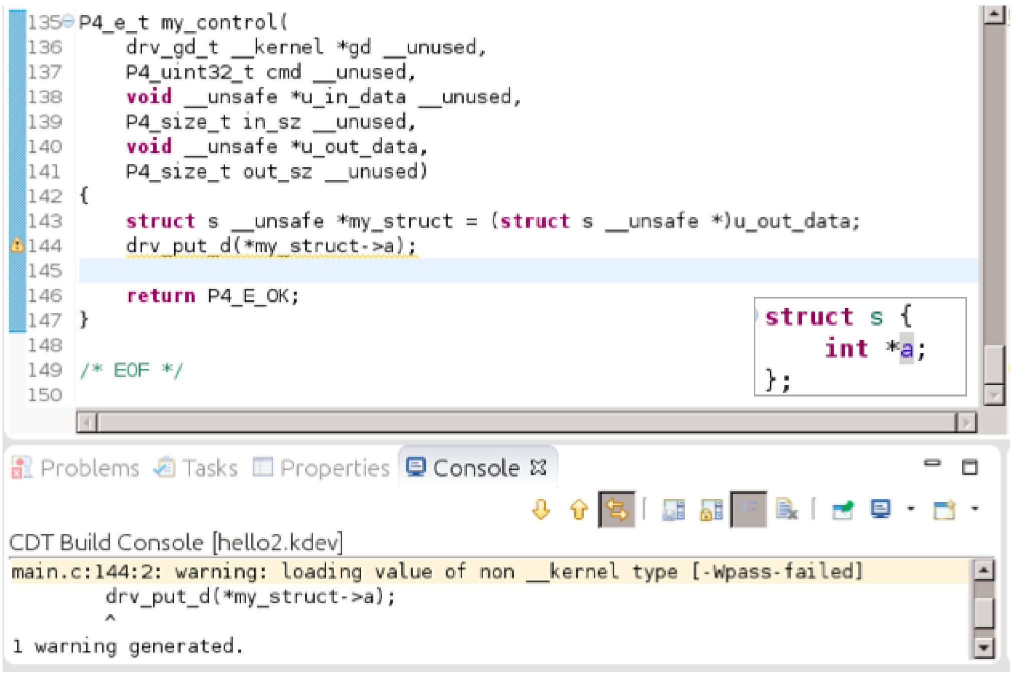
To prevent direct accesses to user space, the option ’Direct memory access’ can be enabled, see figure B.13. This option will enable a static analyze which will verify that dereferences of pointers are only performed on those marked *kernel*, see section 5.2.2.4. With the same code example as presented for the dynamic approach, i.e. section B.1, would this option provide warnings during compile time, see figure B.14.



Parameters

Direct memory access	☑ true
Data copied in	☐ false
Invalidate pointers	☐ false
Change of types	☐ false
Integer to pointer	☐ false
Check with external function	☐ false
Check by inserting instructions	☐ false

Figure B.13: Enable LLVM pass which checks that direct memory accesses are only performed with pointers marked kernel



```
P4_e_t my_control(
    drv_gd_t __kernel *gd __unused,
    P4_uint32_t cmd __unused,
    void __unsafe *u_in_data __unused,
    P4_size_t in_sz __unused,
    void __unsafe *u_out_data,
    P4_size_t out_sz __unused)
{
    struct s __unsafe *my_struct = (struct s __unsafe *)u_out_data;
    drv_put_d(*my_struct->a);

    return P4_E_OK;
}

/* EOF */
```

```
struct s {
    int *a;
};
```

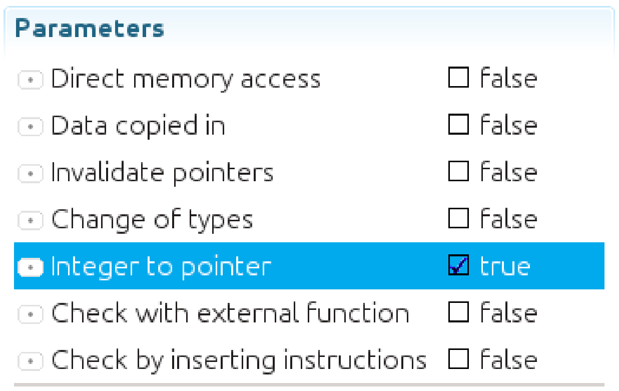
CDT Build Console [hello2.kdev]

```
main.c:144:2: warning: loading value of non __kernel type [-Wpass-failed]
        drv_put_d(*my_struct->a);
        ^
1 warning generated.
```

Figure B.14: Example code which performs a direct access to user space

B.6 Integer to pointer

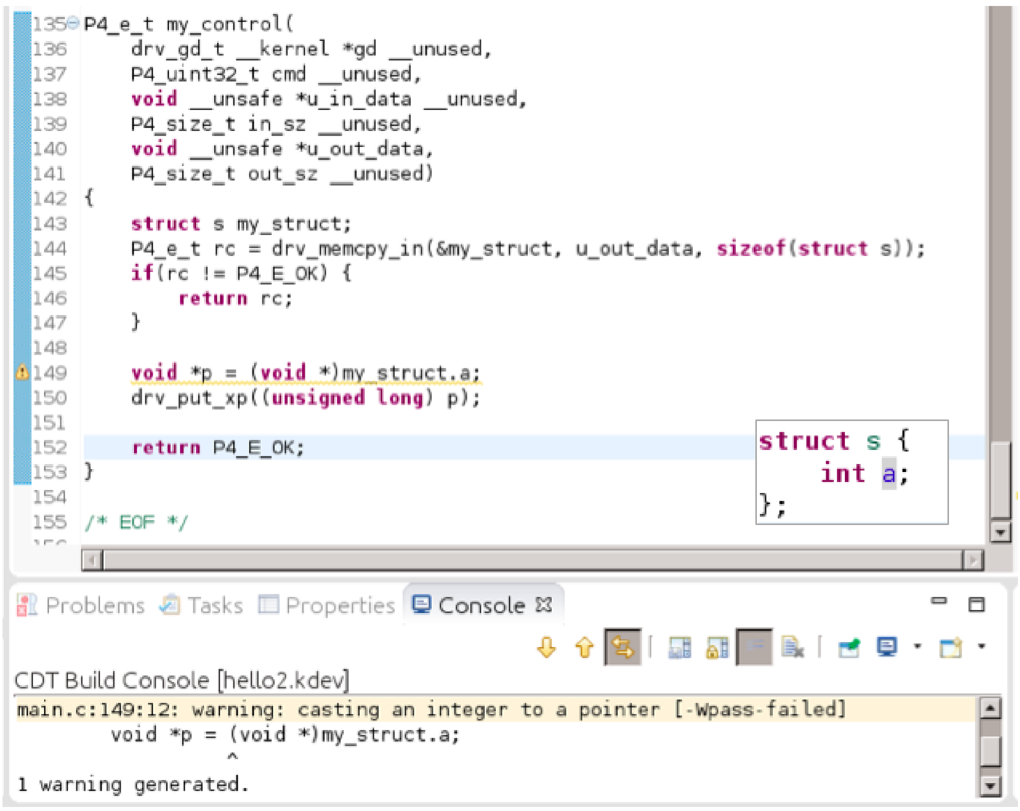
To prevent pointers from being introduced into drivers via cast operations from integers can the option 'Integer to pointer' be enabled, see figure B.15. To illustrate how this will be prevented, the code example presented in figure B.16 will be used. In this figure, a structure that contains an integer enters via a parameter. The structure is then copied in to kernel space, using the PikeOS function *drv_memcpy_in*. Thereafter, the member of the structure is casted from an integer to a void pointer, i.e. introducing a new pointer to the kernel. Since this may introduce serious problems, as explained in section 5.2.2.5, this will generate a warning. If this is desired, then the pointer may be provided with the *unchecked* type which will inform the developer that a new pointer has been introduced that must be checked before its use.



Parameters

Direct memory access	☐ false
Data copied in	☐ false
Invalidate pointers	☐ false
Change of types	☐ false
Integer to pointer	☑ true
Check with external function	☐ false
Check by inserting instructions	☐ false

Figure B.15: Enable LLVM pass which checks that cast from integer to pointer is only performed if the pointer are marked unchecked



```
P4_e_t my_control(
    drv_gd_t __kernel *gd __unused,
    P4_uint32_t cmd __unused,
    void __unsafe *u_in_data __unused,
    P4_size_t in_sz __unused,
    void __unsafe *u_out_data,
    P4_size_t out_sz __unused)
{
    struct s my_struct;
    P4_e_t rc = drv_memcpy_in(&my_struct, u_out_data, sizeof(struct s));
    if(rc != P4_E_OK) {
        return rc;
    }

    void *p = (void *)my_struct.a;
    drv_put_xp((unsigned long) p);

    return P4_E_OK;
}

/* EOF */
```

```
struct s {
    int a;
};
```

CDT Build Console [hello2.kdev]

```
main.c:149:12: warning: casting an integer to a pointer [-Wpass-failed]
        void *p = (void *)my_struct.a;
                  ^
1 warning generated.
```

Figure B.16: Example code which performs cast operation from an integer to a pointer, where the pointer is not marked unchecked

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-06-03

EXAMENSARBETE LLVM-Based Fortification for Kernel Drivers
STUDENT Caroline Brandberg
HANDLEDARE Jonas Skeppstedt (LTH), Henrik Theiling (SYSGO AG, Germany)
EXAMINATOR Flavius Gruian (LTH)

Securing Device Drivers

POPULÄRVETENSKAPLIG SAMMANFATTNING **Caroline Brandberg**

Today's operating systems that are used in highly safety and security critical domains struggle with serious vulnerabilities. This requires new technology, especially concerning the highest error prone software of the operating system, in particular its device drivers.

We live today in a society where it is hard to find anything which is not dependent on technology. Today's cars, trains, airplanes and health-care is highly dependent on software where security and safety are of highest concern.

To be able to ensure people's safety and security one needs a good foundation, namely the operating system. The operating system acts as the layer between the applications and the hardware, hence controlling that everything is done in a correct manner.

Therefore, when dealing with highly safety and security critical domains it is very important to have a certified operating system which can ensure the best base to build from. One problem is that the system can be extended with new software by a third party, where the extended software is referred to as device drivers. These extensions will then be a part of the critical layer, and must therefore be of the same quality as the rest of the system. An extension which does not follow the same standard may introduce serious problems which may crash the whole system.

Device drivers have been shown to have an extraordinarily higher error rate compared to the rest of the system. Since they are integrated to such critical parts of the system it requires a new technology that can prevent these vulnerabilities from being introduced into the system.

There are especially two problems concerning pointers. The first concern is securing the drivers against malevolent users. Pointers received from a destination where a malevolent user can occur must be checked properly. A failure to do so might reveal or destroy critical parts of the system which can cause serious problems. The second concern is securing the extensions from performing memory accesses in a incorrect manner. These problems are very hard to find and test for, hence an introduction of these problems is unfortunately very easy.

To be able to prevent these kinds of problems, we propose a tool which the extended software can use to achieve a high level quality. The tool uses a combination of successful techniques, where we show that only a combination of those can ensure protection against these vulnerabilities. We propose both static analysis and dynamic runtime checks which both have been extended to the Real Time Operating System PikeOS. The statically part of the extension was proven to provide a highly safety net for driver developers, providing valuable information during compile time, whereas the dynamic parts provide almost full error coverage and only showed a very low overhead.