DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor Thesis in Informatics

Dependency Ordering in the Linux Kernel

Paul Heidekrüger

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor Thesis in Informatics

Dependency Ordering in the Linux Kernel

Abhängigkeitsordnungen im Linux Kernel

Author:	Paul Heidekrüger
Supervisor:	Prof. Pramod Bhatotia
Advisors:	Prof. Pramod Bhatotia, Dr. Marco Elver (Google), Charalampos Mainas
Submission Date:	November 15, 2021

Acknowledgments

Above all, I would like to thank my family for their continuous support and encouragement during my studies.

I would like to thank Pramod Bhatotia, Marco Elver and Charalampos Mainas for taking a chance on me with this thesis, being patient and always being on hand for discussions and questions in regular meetings. I really enjoy working on this topic and cannot overstate enough how much I have learnt. I am incredibly thankful for the opportunity.

Many thanks to Rodrigo Rocha for always having an open ear for LLVM-related questions, many thanks to Jörg Thalheim for equipping me with the required computing power and generously helping with NixOS-related questions.

A thank you also goes to Sophia Adelmeier for helping with all administrative questions regarding my work.

And finally, a nod to Vincent Picking for productive bachelor-thesis co-working sessions.

In memory of A.H. and K.H.

Abstract

When relevant papers are titled *'Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel'* [1], it should not surprise that Linux kernel development often goes a non-standard way of its own. Mainly, this is a result of wanting to meet the goal of peak performance. One example of this is the Linux kernel memory model which differs ever so slightly from the C11 memory model.

Over the last couple of years, the Linux kernel community has become increasingly worried about the chance of compilers, which respect the C11 memory model, introducing optimisations which break certain dependencies the Linux kernel relies on being preserved as per its own memory model.

The potential of this happening has been extensively talked about on the Linux kernel mailing list, but still, discussions have left a real-world example of a dependency ordering being broken in the Linux kernel to be desired.

We present a means for identifying dependency orderings in the Linux kernel before compiler optimisations run and checking whether the code was optimised such that a dependency was lost. Our approach consists of two compiler passes which can be run when building the Linux kernel with the LLVM Clang compiler.

Our work has already led to a contribution to the Linux kernel mailing list, and we are confident that based on this work, we will be able to provide a reliable mechanism for identifying (broken) address and control dependencies in the Linux kernel.

Contents

1 Introduction

When researching concurrency, one will quickly enter the realms of what Linus Torvalds considers the rocket science of computer science [2]. Oh no.

However, although not intended, it appears to be a fitting description for Linux kernel development, which, like rocket construction, often faces problems of a nature common standards cannot address to the engineers' satisfaction. One instance where the Linux kernel, quite literally, deviates from common standards is its memory consistency model - the Linux Kernel Memory Model (LKMM) - which governs the behaviour of concurrent code when shared memory is accessed. Differences to the C-language memory consistency model are subtle [3], but with compiler optimisations becoming increasingly sophisticated, there exists a chance of optimisations leading to undesired and hard-to-debug behaviour in Linux kernel code.

A concrete case where the differences in memory consistency models could lead to bugs is that of a read -> read or read -> write address dependency. The Linux kernel relies on such accesses to shared memory being ordered by the architecture. One, granted, artificial example that has been discussed among Linux kernel developers is that where an address dependency could be transformed into a control dependency because the compiler was able to infer some property which the result of the first load might have [4]. Such a transformation is shown in 1.1 and 1.2.

By converting the address dependency in 1.1 into a control dependency such as in 1.2, a weakly ordered architecture - such as arm64 or Power - could speculate the if branch before *x = READ_ONCE(*foo)* becomes observable, making *bar* available and allowing *y = READ_ONCE(*bar)* to be speculatively executed, thereby breaking dependency ordering.

In fact, the transformation of address to control dependencies is not necessary for reordering to become a problem. Compilers might apply similar optimisations for

```
x = READ_ONCE(*foo);
bar = &x[42];
y = READ_ONCE(*bar);
```

Figure 1.1: A simple address dependency from y to x [4]

```c
x = READ_ONCE(*foo);
if (x == baz)
    bar = &baz[42];
else
    bar = &x[42];
y = READ_ONCE(*bar);
```

Figure 1.2: A transformation of 1.1 into the above control dependency would break the desired instruction ordering [4]

```c
#define MAX 1

x = READ_ONCE(*foo);
if (x % MAX == 0)
    WRITE_ONCE(*bar, 1);
```

Figure 1.3: A control dependency with a constant conditional branch, giving way to optimisations which could remove conditional branching [6]

existing read -> write control dependencies, breaking the conditional dependency between the two memory accesses. Read -> read control dependencies are generally not ordered and therefore not considered [5].

Again, the artificial example in 1.3 shows how such a transformation could look: the write is dependent on the read through the if condition. Since the condition evaluates to a constant, a compiler could optimise it away, leaving no more dependency between the read and the write and allowing architectures to reorder the memory accesses. At several occasions [7] [8] [4], the community has expressed the need for a tool to identify such broken dependency orderings with the value proposition being twofold. Firstly, such a tool would help finding and addressing broken dependencies in the Linux kernel, and secondly, it would bring the additional advantage of being able to present concrete cases of broken dependencies in the Linux kernel to the C standard committee as a base for discussing C memory consistency model revisions in favour of LKMM. That compilers break control dependencies as shown in 1.3 has been demonstrated and discussed in the context of LLVM [9]. However, there exists no mechanism to identify broken dependencies as part of compilation.

This thesis presents two LLVM passes, which are able to identify a subset of broken dependencies when building the Linux kernel with the LLVM Clang compiler.

2 Background

In the following, we cover memory consistency models, LLVM and the potential problem of broken dependency orderings in the Linux kernel. If possible, we follow a top-down approach for the levels of abstraction.

2.1 Introducing Memory Consistency Models

Computer science has come up with abstractions, namely memory consistency models (MCMs), commonly referred to as memory models, to reason about concurrent code and make sense of the reordering of instructions done by modern compilers and architectures. To borrow a definition from [10]:

> [A Memory Consistency Model] formally specifies how the memory system will appear to the programmer. Essentially, a memory consistency model restricts the values that a read can return.

The Linux kernel documentation [5] outlines two additional intuitions. One, a memory consistency model can be used to verify that, given a piece of code and a set of values, a given outcome is or is not possible. Two, a memory consistency model can be understood as an oracle, which, given a piece of code, can predict potential outcomes. Both intuitions are equivalent. Different MCMs assure different levels of consistency. MCMs are loosely categorised by the restrictions they place on reordering, where a stronger MCM leaves less room for reordering than a weaker MCM.

2.1.1 Sequential Consistency

One of the stronger and more intuitive guarantees any MCM can make is that of sequential consistency. Its definition goes back to [11] and states the following.

> The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

```
x = 0, y = 0

T1() {
    STORE(4, x)
    yp = READ(y)
}
T2(){
    STORE(2, y)
    xp = READ(x)
}
```

Figure 2.1: An abstract example where reordering can become a problem

Given the abstract example in 2.1, a sequentially consistent execution path would be to execute the two threads one after another. Starting with T1, this would lead to the outcome *(xp=4, yp=0)*. Another sequentially consistent execution could be a round-robin alternation between T1 and T2, executing one instruction at a time. Starting with T1, this would lead to the outcome *(xp=4, yp=4)*. The outcome *(xp=0, yp=0)* cannot be the result of a sequentially consistent execution as it implies that both *READ* instructions ran before the corresponding *STORE* instructions to x and y respectively, thereby not adhering to the programme order. In this example, if a value is written by a thread, it becomes immediately visible to all other threads. [12]

2.1.2 Relaxing Memory Consistency

Sequential consistency is not desirable when running performance-critical code as it can impose unnecessary restrictions on reordering. 2.2 modifies 2.1 such that requiring the execution of *T2* to be sequentially consistent would not be necessary as it would not affect its result. In *T1* on the other hand, the *READ* instruction now depends on the preceding *STORE* instruction. Ordering is therefore desired.

Considering the 'is succeeded by' relation on the programme order as shown in 2.3, we end up with four possible orderings which can be preserved: $r \rightarrow_{succ} r$, $r \rightarrow_{succ} w$, $w \rightarrow_{succ} r$ and $w \rightarrow_{succ} w$. We can relax memory consistency by allowing more outcomes to occur. For instance, permitting non-dependent writes to be ordered ahead of reads would break $w \rightarrow_{succ} r$. In 2.1, this would enable the outcome *(xp=0, yp=0)*. We will see that this in fact common practice for modern computing architectures.

```
x = 0, y = 0, z = 0

T1() {
    STORE(4, x)
    yp = READ(y)
}

T2(){
    STORE(2, y)
    zp = READ(z)
}
```

Figure 2.2: A modified version of 2.1 where a sequentially consistent execution would not be required

$$x, y \in \{r, w\} : x \rightarrow_{succ} y \iff x\ comes\ before\ y\ in\ programme\ order$$

Figure 2.3: The 'is succeeded by' relation

2.2 Memory Consistency Models on the Language Level

An MCM may apply to either the architecture or source level in the abstract computing stack. Both are strongly intertwined as toolchains must respect the source-level MCM when compiling source code for a target architecture with an MCM of its own. Compilers then insert, if necessary, memory barriers to make the code behave as intended by the programmer when compiling for a supported target architecture.

2.2.1 The C Memory Consistency Model

Concurrency, in the form of threads and atomics, became a part of the C programming language with the introduction of the C11 standard in 2011. As C does not guarantee atomicity for all operations on variables from C11 onwards - this generally depends on the architecture - using the atomics library is non-negotiable for avoiding data races in concurrent C code. Programmers can provide memory order arguments to memory accesses, specifying how atomic and non-atomic memory accesses are ordered against atomic memory accesses. C11 and beyond support five different memory orders, ranging from *memory_order_relaxed*, which imposes no limits on reordering, leaving the execution order up to compilers and the architecture, to *memory_order_seq_cst*, guaranteeing a sequentially consistent execution [13]. If not

memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_seq_cst

Figure 2.4: C11's memory orders [13]

specified otherwise, *memory_order_seq_cst* is attached to atomic operations in C by default, sacrificing performance in favour of preventing unexpected outcomes.

Of specific interest to us are *memory_order_release*, *memory_order_consume* and *memory_order_acquire*. Where the former, being attached to an atomic write, is often paired with one of the latter, being attached to an atomic read from the same variable.

Release-Acquire Guarantees

memory_order_release can be passed to an atomic store *s* in a thread *T1* as an optional argument, guaranteeing that all load and store operations that come before *s* in a given thread also happen before *s* at runtime. We observe that this does not pose any additional restrictions on the loads and stores before *s*, nor does it prevent load and store operations which come after *s* in programme order from being observable before *s* at runtime. *memory_order_acquire* poses similar restrictions, only this time, it is attached to an atomic read *r* in thread *T2*. It guarantees that no reads and writes that come after *r* in the programme order of *T2* become observable before *r* at runtime. Again, this does not exclude load or store operations that come before *r* in programme order from becoming observable after *r* at runtime. It is common to use release and acquire orders together, guaranteeing that all instructions starting at *r* in *T2* are able to observe the results of all memory accesses that happen up to *s* in *T1*.

Addressing the Elephant in the Room - *memory_order_consume*

memory_order_consume is identical to *memory_order_acquire* except that it only affects dependent memory accesses. Instead of all loads and stores, now only those which depend on the *memory_order_consume* operation are constrained. Notice that acquire semantics imply consume semantics; if all accesses are constrained, then the dependent accesses are constrained. One shortcoming of current C compilers is that they make use of this convenience by promoting every *memory_order_consume* to a *memory_order_acquire*, thereby leaving performance optimisations on the table as additional constraints are imposed which the programmer explicitly wanted to avoid. [13]

2.3 Memory Consistency Models on the Architecture Level

Whilst language-level memory models are more generic and architecture-independent, architecture-level memory models are concerned with the instruction order at runtime.

2.3.1 Total Store Ordering

The strongest consistency guarantee one can encounter in today's computing architectures is that of Total Store Ordering (TSO), which appears in X86 or SPARC architectures [14] [15]. TSO guarantees that values are always written to memory in the order in which the store instructions ran in the corresponding thread and that every thread always has access to its most recent store when loading from the same memory location. However, it places no guarantee on the order of memory loads from different addresses in relation to writes and when writes to memory become visible to other threads. It achieves this by maintaining a FIFO queue for every thread, which holds values that ought to be written to memory. The FIFO property of the queue guarantees that values are written in the order in which they are executed and that they become visible to all other threads at the same time. When loading a value, a thread checks its FIFO queue first before accessing memory. This means that every thread has its own view of the current memory state. That is, a unification of the values in memory and the writes performed by the given thread thus far. [12] has abstracted TSO's properties such that it can be visualised as in 2.5. In particular, TSO allows for the outcome (xp=0, yp=0) in 2.1.

Reordering is implicitly constrained if instructions depend on each other. For example, read -> read or read -> write data dependencies, this includes address dependencies, are not reordered by x86 or any architecture supported by the Linux kernel for that matter, except DEC Alpha. [5]

2.3.2 Weak Memory Consistency

Weaker MCMs, such as those of Arm or IBM Power architectures, take reordering a step further, placing hardly any constraints on reordering of reads and writes. [12] abstractly describe this as every thread having its own copy of memory. In contrast to TSO, instructions can even be speculatively executed before previous conditions have been fully resolved. This can become a problem as shown in 1. However, there still exist mechanisms for implicitly imposing ordering on an architecture level. For example, through data dependencies. [12]

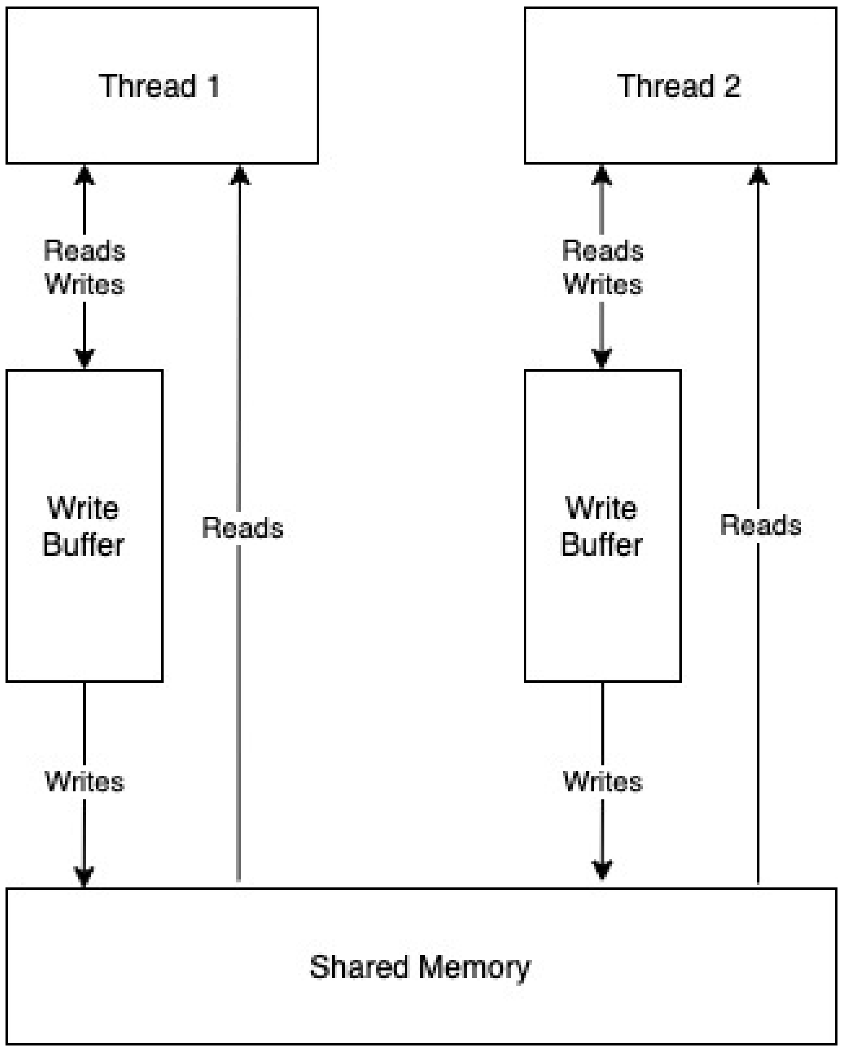


Figure 2.5: Abstractly visualising a strong MCM: TSO on a two-thread system

2.3.3 Ambiguities in Memory Consistency

Even though architecture-level memory models are as close to the executed code as feasibly possible, hardware vendors' specifications still leave room for ambiguities. In the case of TSO, this lead to the rigorous formalisation of x86-TSO in [16].

2.3.4 Mechanisms That Influence Reordering on the Architecture Level

In an ideal world, where loads and stores to memory happen instantaneously, no reordering of instructions would be needed and sequentially consistent executions would be the default. Since CPU speeds have increased exponentially faster than memory access speeds over the last decades, vendors had to come up with increasingly sophisticated mechanisms for reducing the rising costs, i.e. CPU cycles, of memory accesses. Amongst others, the Armv8-A reference manual [17] describes the following:

- Multiple instructions can be issued by CPUs within the same cycle, allowing true concurrency and potentially out-of-programme-order executions.
- As shown in 2.2, non-dependent instructions can be executed out of order to ensure the best utilisation of computing power.
- Instructions can be speculatively executed before control flow has been fully resolved.
- Loads and stores to memory can be batched together for reducing latency.

Only in multi-threaded programmes this can lead to unexpected behaviour. In single-threaded programmes, an execution in programme order is guaranteed.

2.4 The LLVM Project

2.4.1 The Philosophy of LLVM

The name LLVM unifies several different projects under one umbrella. Most prominently, LLVM includes the Clang compiler, but it also offers a linker, a debugger and an implementation of the C++ standard library. In short, it provides a whole toolchain of its own for going from source code to executable, which was specifically designed with reusability in mind. Reusability in LLVM is achieved through high modularity. As a result, real-world uses of LLVM are very broad and range from gaming consoles, to operating systems and image processing [18]. The LLVM project of particular interest for this thesis is the Clang compiler. [19]

```
"/home/paul/src/DoitLK-llvm/build/bin/clang-13" "-cc1" "-triple" "x86_64-
    ↪ unknown-linux-gnu" "-emit-obj" "-mrelax-all" "--mrelax-relocations"
    ↪  "-disable-free" "-main-file-name" "helloworld.cpp" "-mrelocation-
    ↪ model" "static" "-mframe-pointer=all" "-fmath-errno" "-fno-rounding
    ↪ -math" "-mconstructor-aliases" "-munwind-tables" "-target-cpu" "x86
    ↪ -64" "-tune-cpu" "generic" "-debugger-tuning=gdb" "-fcoverage-
    ↪ compilation-dir=/home/paul/src/linux" "-resource-dir" "/home/paul/
    ↪ src/DoitLK-llvm/build/lib/clang/13.0.0" "-internal-isystem" "/home/
    ↪ paul/src/DoitLK-llvm/build/bin/../include/c++/v1" "-internal-
    ↪ isystem" "/home/paul/src/DoitLK-llvm/build/lib/clang/13.0.0/include
    ↪ " "-internal-isystem" "/usr/local/include" "-internal-externc-
    ↪ isystem" "/nix/store/q141hd8jl7in5223jmf7kmx9h517km4p-glibc
    ↪ -2.32-54-dev/include" "-fdeprecated-macro" "-fdebug-compilation-dir
    ↪ =/home/paul/src/linux" "-ferror-limit" "19" "-fgnuc-version=4.2.1"
    ↪ "-fcxx-exceptions" "-fexceptions" "-fcolor-diagnostics" "-faddrsig"
    ↪  "-D__GCC_HAVE_DWARF2_CFI_ASM=1" "-o" "/run/user/1007/helloworld-0
    ↪ c0a09.o" "-x" "c++" "helloworld.cpp"
[...]
```

Figure 2.6: A part of the Clang frontend command when compiling a simple *helloworld.cpp* file

2.4.2 An Overview of Clang

Clang sticks to LLVM's philosophy of modularity and can once again be dissected into several subcomponents. When users interact with Clang on the command line, they are invoking its driver component. The driver abstracts from the operation system, providing a unified interface for running Clang on any supported system.

The '-###' driver option prints out the arguments which the driver eventually passes to the frontend. For example, for a simple *helloworld.cpp* programme, *'clang++ helloworld.cpp -o helloworld'* might expand to something to 2.6. The frontend only marks the beginning of the several stages the source code takes until it ends up in an executable. Clang can print out the different stages a programme goes through with the *'-ccc-print-phases'* option. We explain them briefly [20]:

Preprocessing As part of preprocessing the source code, macros are expanded, header files included, comments removed and other preprocessor commands resolved until eventually, a translation unit is generated.

Parsing and Semantical Analysis The parsing stage transforms translation units into a tokenised form. For example, a simple assignment such as *′x = 42′*, is broken up into its left-hand and right-hand sides. This stage can also generate many familiar compiler warnings. For *′double foo = ″bar″′*, it might warn about type mismatch and a missing semicolon for example. If no error is thrown, an abstract syntax tree (AST) is generated, which marks the final step before generating the IR.

Code Generation and Optimization The AST is used for generating IR, which is then optimised and translated into assembly code for the specified target - typically in a *′.s′* file. This is the stage our work is concerned with.

Assembler The assembler takes the output from the previous stage and assembles it into an object file.

Linker The final stage eventually links together all relevant object files into an executable.

How much freedom Clang can take in its optimisations is determined by the *′-O′* flag passed to the driver. For example, *′-O0′* disables optimisations and *′-O2′* enables most optimisations and is what we use to build the Linux kernel [21]. For optimising code, Clang uses LLVM's pass infrastructure. Passes may run on modules, functions or loops for example and are categorised into transformation passes, which modify the code, e.g. by optimising it, and analysis passes, which obtain information from the code. Clang's use of passes again underlines the modularity of LLVM. Not only can the passes be language independent, but they are decoupled from the projects they are used by. This means one can easily extend Clang using LLVM passes without specific knowledge of Clang's internals.

2.4.3 LLVM Intermediate Representation

LLVM IR is characterised by Single Static Assignment (SSA) form, guaranteeing that each variable is assigned a value exactly once. Whilst it may look cumbersome, it makes optimisations easier. Note that IR is, like C and C++, platform dependent. Differences become apparent when using *′sizeof(long)′* for instance as its result depends on the architecture [22].

On an IR level, it may appear that the value on the left-hand and right-hand side of an assignment are different, i.e. we are assigning one value to another value. With SSA properties however, this assignment can only happen once, meaning that every register can be identified by its value - registers are their own value. In 2.7 for example, *′%0′* is *′%0 = load i32, i32* %tmp, align 4, !dbg !82′*. The *′=′* is more akin to equality in the mathematical sense.

```
; Function Attrs: noinline nounwind null_pointer_is_valid optnone
    ↪ sspstrong
define internal void @lkm_exit() #0 !dbg !76 {
entry:
  %tmp = alloca i32, align 4
  store i32 0, i32* %tmp, align 4, !dbg !79
  %0 = load i32, i32* %tmp, align 4, !dbg !82
  ret void, !dbg !83
}
```

Figure 2.7: A simple IR function

IR instructions are grouped into basic blocks which are linked together to form the control flow graph of a function. One or several functions make up a module, i.e. translation unit.

One mechanism for passing information through the optimisation pipeline is annotating IR instructions with metadata. This is further discussed in 6.

2.5 Potentially Broken Dependency Orderings in the Linux Kernel

Above, we have provided an overview of the different factors influencing compilation of concurrent C programmes, i.e. the C memory model, compiler optimisations and various architecture-level memory models of different strengths. The Linux kernel makes an important change here, as it maintains a language-level memory model of its own.

2.5.1 The Linux Kernel Memory Model

Since the Linux kernel community has been using atomics and memory orderings, e.g. *release-acquire*, before they were introduced in the C standard, the community had to come up with its own implementation atomics in C. Once atomics officially became a part of the C language with C11, it would seem natural for the Linux kernel to switch away from its custom implementation. In fact, this has been proposed on the Linux Kernel Mailing List (LKML), but has been met with doubts. One concern was the lack of a *memory_order_consume* implementation with current compilers [23], which could have significant effects as the Linux kernel's widely-used implementation of *rcu_dereference()* depends on *memory_order_consume* semantics. [3] With the Linux

```
int a[20];
int i;

r1 = READ_ONCE(i);
r2 = READ_ONCE(a[r1]);
```

Figure 2.8: An address dependency as per [5]

```
int x, y;

r1 = READ_ONCE(x);
if(r1)
    WRITE_ONCE(y, 1984);
```

Figure 2.9: A control dependency as per [5]

kernel often facing problems that were outside of the capabilities of the C-standard, the above being one example, the community decided early on to maintain a matching memory consistency model for its atomics - the Linux Kernel Memory Consistency Model (LKMM).

The LKMM is loosely defined in [5] and formally in [1], differing from the C memory consistency model in subtle points [3]. It abstracts from C source code into *events* which are represented as macros and resolved by the preprocessor. A memory access through one of the macros is referred to as *annotated*. LKMM maintains three kinds of events for accessing shared memory, each of which associates with different annotations [5]:

Read Events Read events correspond to loads from shared memory, such as calls to *READ_ONCE()*, *smp_load_acquire()*, or *rcu_dereference()*

Write Events Write events correspond to stores to shared memory, such as calls to *WRITE_ONCE()*, *smp_store_release()*, or *atomic_set()*

Fence Events Fence events correspond to memory barriers (also known as fences), such as calls to *smp_rmb()* or *rcu_read_lock()*

Several relations link different events. For instance, address and control dependencies are defined as follows with 2.8 and 2.9 being provided as examples [5]:

Address Dependency A read event and another memory access event are linked by an address dependency if the value obtained by the read affects the location accessed by the other event. The second event can be either a read or a write.

Control Dependency A read event and another memory access event are linked by a control dependency if the value obtained by the read affects whether the second event is executed at all.

In the case of address dependencies, LKMM relies on the fact that they are ordered by nearly all architectures. In order for code to arrive at the architecture however, it has to go through the compilers' optimisation pipelines which the Linux kernel community fears could be a problem as the Linux kernel does not use the atomics and memory orderings specified by the C standard, which compilers take into account. Most prominently, the community is concerned about address and control dependencies being affected as shown in 1. Most recent discussions were a result of a proposal for Clang Link-Time Optimisation (LTO) support on arm64 [24]. As this issue is yet to be explored in detail and the community is still unclear on whether this is a problem worth addressing, the convenience of a tool was discussed which would be able to detect such broken dependencies [7] [8] [4]. We propose such a tool as part of this thesis.

2.5.2 Clang and the Linux Kernel

Whilst the standard approach would be to compile a Linux kernel with GCC, it does support compilation with Clang out of the box [25]. In fact, there exist several Linux kernel derivatives, such as Android, which can only be compiled with Clang. Since the topic of broken dependencies has been discussed in the context of Clang-built kernels and LLVM LTO support, due to the modular and extensible nature of LLVM, we choose to implement the tool as an extension to the Clang compiler in the form of compiler passes.

3 System Overview

In the following, we outline a means for identifying a subset of dependency orderings in the Linux kernel, tracking them through compiler optimisations and providing a list of broken dependencies to users. We implement the tool as an extension to the LLVM Clang compiler in the form of compiler passes, where the broken dependencies are printed as warnings to the *stderr* stream. This allows a natural integration in the build process of the Linux kernel, as Clang is supported out of the box [25]. We provide two passes. One for annotating the dependencies before compiler optimisations and one for verifying the annotated dependencies after compiler optimisations. Both passes are inserted into the Clang pipeline accordingly. Being an extension to LLVM, the compiler passes annotate and verify dependencies on the level of LLVM intermediate representation (IR). We test the tool with a kernel module, containing cases of dependencies that we deem relevant. Dependencies are artificially broken after they have been annotated by the first pass and are then detected as such by the tool after compiler optimisations have run when building a Linux kernel with modules enabled. Results are discussed in 6. Currently, our tool identifies read -> read and read -> write address dependencies where both accesses are annotated as per LKMM. Further development is discussed in 9.

4 Design

4.1 Source-Level Dependency Orderings in LLVM IR

The proposed compiler passes annotate address dependencies. As per 2, address dependencies begin with *READ_ONCE()* (*R*) or another annotation which eventually expands to a *R* access and end with *R*, *WRITE_ONCE()* (*W*) or another annotation which eventually expands to *W*. 4.1 and 4.2 shows how the *READ_ONCE()* and *WRITE_ONCE()* macros are defined in *linux/include/asm-generic/rwonce.h*. We observe that *R* and *W* eventually map to a volatile pointer. As annotation and verification happen on the LLVM IR level, we require a unique representation of *R* and *W* in LLVM IR. Since he LLVM Language Reference guarantees that volatile accesses carry down to IR [26], the Linux kernel requires accesses to shared memory to be annotated and *R* and *W* expand to a volatile pointer, we make the following assumption: all memory accesses marked with *volatile* in LLVM IR are the result of a shared memory access in Linux kernel source code and can therefore be part of a relevant dependency. We accept potential false positives as a result of source code not adhering to the kernel standard. Therefore, in IR, address dependencies are characterised by a first load instruction with attached *'volatile'* as well as a second load or store instruction with attached *'volatile'* which loads from or stores to an address that depends on the first volatile load.

To track dependencies between values, we introduce the notion of a dependency

```
#define READ_ONCE(x)       \
({        \
       compiletime_assert_rwonce_type(x);    \
       __READ_ONCE(x);      \
})
```

```
#define __READ_ONCE(x) \
(*(const volatile \
__unqual_scalar_typeof(x) *)&(x))
```

Figure 4.1: READ_ONCE() in the Linux kernel [27]

```
#define WRITE_ONCE(x, val)      \
do {          \
       compiletime_assert_rwonce_type(x);    \
       __WRITE_ONCE(x, val);      \
} while (0)
```

```
#define __WRITE_ONCE(x, val)      \
do {          \
       *(volatile typeof(x) *)&(x) = (val);    \
} while (0)
```

Figure 4.2: WRITE_ONCE in the Linux kernel [27]

$$\begin{aligned} D_v &= \{v' \in V \mid \forall v' : reaches(v, v') \ \wedge \ depends(v', v)\} \\ reaches(v, v') &\iff control\ flow\ can\ reach\ v'\ from\ v\ for\ v,\ v' \in V \\ depends(v', v) &\iff v\ uses\ v' \vee \exists v'' \in V : depends(v', v'') \end{aligned}$$

Figure 4.3: Defining a dependency chain

chain as shown in 4.3. A dependency chain is started by a volatile load and contains all the values that depend either directly or indirectly on the value of the volatile load. We think of it as a chain which runs from the start to the end of a dependency, where the links are the depending instructions. For a volatile load instruction with value v as part of the set of all values V in the module, we define the dependency chain as a set D_v in 4.3. Per our definition, a dependency chain might contain several dependencies which all start at the same load.

4.2 Algorithmic Approach

We propose an algorithm for annotating and verifying address dependencies in LLVM IR based on the Breadth-First Search (BFS) outlined in [28]. For every function in a module, the algorithm runs on the control flow graph (CFG) of the function's basic blocks (BBs) and annotates / verifies dependencies as it iterates over the instructions in a BB. Furthermore, it carries out interprocedural analysis, meaning that it can annotate / verify dependencies across function boundaries. The proposed algorithm is one-pass in the sense that it only looks at BBs as often as they appear in paths through the CFG. This does not prohibit the algorithm from looking at the same BB multiple times. For example, in the case of a function call, the algorithm looks at the function itself and then looks at it again when following the corresponding function call in another function.

4.3 The Annotation Pass

The annotation pass is called for every IR module which is being compiled. This happens before most of the optimisation passes run and is discussed further in 5.

The pass begins by iterating over every function in the module in arbitrary order, skipping those that are empty. Once it finds a function that is not empty, it initialises two empty maps, *beginnings* and *endings*, which hold the annotations to be made for beginnings and endings respectively, as well as the function's post-dominator tree which is used for ruling out conditional paths in the CFG. Both *beginnings* and *endings* map unique IDs to dependency beginnings / endings respectively. Their representation is discussed in 5. The pass then starts the modified BFS on the function's CFG. The BFS maintains a context which contains all relevant data structures for annotation. Once the search has returned, the pass proceeds with annotating the identified dependencies and finishes by artificially breaking some of the dependencies if it is currently looking at one of the functions in our testing module.

4.4 The Verification Pass

Like the annotation pass, the verification pass is called for every module being compiled, only this time after most of the optimisations have run. Again, this is further discussed in 5.

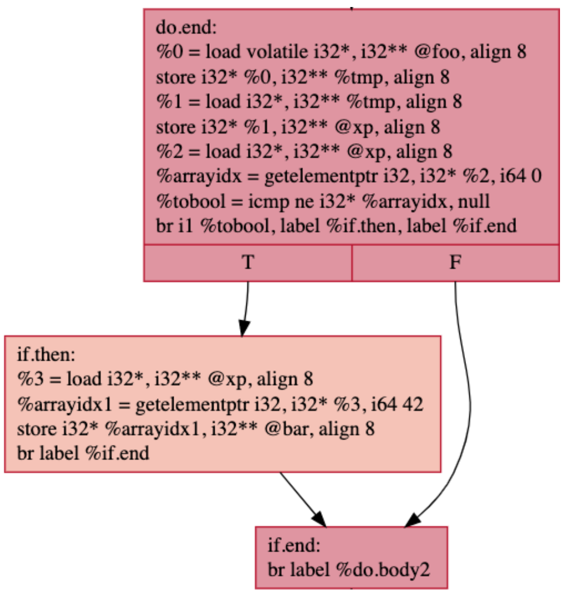
Before iterating over all the functions in the module, the verification pass initialises an empty set, which holds the IDs of the dependencies that have already been verified, as well as *beginning* and *ending* maps. Like the annotation pass, the verification pass iterates over all functions in the module, skipping those that are empty. After running a BFS on all non-empty functions, the pass finishes by printing all broken dependencies if it was able to find any.

4.5 Breath-First Search for Annotation and Verification of Dependencies

The BFS function has been abstracted enough to be used for annotation as well as verification. it is called for non-empty function in a module and orchestrates the traversal of the CFG. 1 shows a high-level version of the BFS in pseudo code.

Algorithm 1 A modified BFS for identifying (broken) dependencies

```
1: procedure BFS($context$)
2:     $bfs\_queue \leftarrow [(context.BB, context.potential\_beginnings)]$
3:     $visited\_BBs \leftarrow \emptyset$
4:     while $bfs\_queue \neq []$ do
5:         $update\_context()$
6:         $new\_potential\_beginnings \leftarrow checkBBForDeps(context)$
7:         if $context.inter\_procedural\_level > 0$ then
8:             $handle\_block\_with\_return\_inst()$
9:         end if
10:        for all $s \in successors(context.BB)$ do
11:            if $s \in visited\_BBs$ then
12:                $handle\_duplicate\_BB()$
13:            else
14:                $bfs\_queue.push\_back(\{s, new\_potential\_beginnings\})$
15:                $visited\_BBs \cup \{s\}$
16:            end if
17:        end for
18:    end while
19: end procedure
```



```
do.end:
%0 = load volatile i32*, i32** @foo, align 8
store i32* %0, i32** %tmp, align 8
%1 = load i32*, i32** %tmp, align 8
store i32* %1, i32** @xp, align 8
%2 = load i32*, i32** @xp, align 8
%arrayidx = getelementptr i32, i32* %2, i64 0
%tobool = icmp ne i32* %arrayidx, null
br i1 %tobool, label %if.then, label %if.end
```

```
if.then:
%3 = load i32*, i32** @xp, align 8
%arrayidx1 = getelementptr i32, i32* %3, i64 42
store i32* %arrayidx1, i32** @bar, align 8
br label %if.end
```

```
if.end:
br label %do.body2
```

Figure 4.4: A single *'if.then'* BB which must be looked at before the *'if.end'* BB in the BFS

The BFS initialises the following:

- A queue, containing all BBs which are yet to be looked together with a state of all dependency chains from their immediate predecessors. It is initialised with the entry block of the function and an empty map of instructions to potential beginnings, including their dependency chains. In the following this map is referred to as DepChainMap
- A set, containing all BBs that have already been visited

Once initialised, the BFS function iterates over the queue until it is empty. In the loop, it pops the first element from the queue - a pair consisting of a BB and a DepChainMap - and call the *checkBBForDeps()* function, passing it the current context. *checkBBForDeps()* eventually returns the new DepChainMap which holds after control flow runs through the current BB. The BFS now needs to check in which context it is being called. Callees could be either the annotation pass, the verification pass or the *checkCallForDeps()* function, which is responsible for interprocedural analysis. In the latter case, the BFS needs to check whether the BB it just looked at is terminated with a return instruction. If yes, it needs to adjust the return set in the current context accordingly. This is touched on again in the following subsections. In short, the pass checks if the return value of a function is part of any dependency chain. The loop is completed by adding new elements, i.e. successors of the current BB, to the queue, taking into account the possibility of duplicates. Succeeding BBs are partitioned according to post dominance. Every succeeding BB which does not post dominate the current BB is looked at before every succeeding BB which does post dominate the current BB. This is needed to maintain the breath-first nature of the search as illustrated by 4.4. If the BFS were to look at the *'if.end'* BB before *'if.then'* BB, the BFS function would not respect its breath-first requirement. If a succeeding BB is not present in the queue, we can simply add it to the queue together with the current DepChainMap. However, multiple BBs might have the same successor. Therefore, if the BB is already present in the queue, we need to update the BB's DepChainMap accordingly. We do this by intersecting the two dependency chains. Only values which are present in both dependency chains are be preserved. A requirement which is be further discussed in 6. If a succeeding BB has been visited before, it is skipped as this implies a cycle.

4.6 Building the Dependency Chains

The *checkBBForDeps()* function, which is called by the BFS, is responsible for tracking the dependency chains. It iterates over all instructions in the current BB, handling the different instruction types accordingly and adding their values to the correct dependency chains.

4.6.1 Handling Volatile Loads and Stores

Every time a volatile load or store instruction is encountered, a dependency can potentially be annotated or verified. In the case where the ending of a dependency is encountered, *checkBBForDeps()* checks for post-domination, i.e. if the BB of the second memory access post-dominates the BB of the first memory access. A basic block *BB2* is said to post-dominate a basic block *BB1* if every path from *BB1* to the end of the function must go through *BB2* [29]. This guarantees that the second load is not optional when control flow goes from the first load to the end of the function. It is one step in ruling out control dependencies from being falsely annotated as address dependencies and is further discussed in 6. The check for post-dominance accounts for the two memory access occurring in different functions.

4.6.2 Interprocedural Analysis

Every time *checkBBForDeps()* encounters a call, it differentiates between the following cases:

Case 1: Do dependencies start in the called function and end in the called function?

Case 2: Do dependencies start in the called function and end in the calling function?

Case 3: Do dependencies which begin in the calling function end in the called function?

Case 4: Do dependencies which begin in the calling function run through the called function and end in the calling function?

The depth of interprocedural analysis is statically limited by a constant defined in source code. Therefore, *checkCallForDeps()* immediately checks the current depth after being called. Only if it is under the limit, it starts a new BFS in the called function. For covering cases 3 and 4, *checkCallForDeps()* (and therefore the BFS) receive the dependency chains, which contain the functions arguments, as part of the context. Cases 1 and 2 are handled as they would be without interprocedural analysis. The BFS carries on as described above. When encountering a BB which is terminated with a return instruction however, some additional actions are required. Since the BFS is carrying out interprocedural analysis and it has encountered a return value, to cover cases 2 and 4, it has to check whether the return value is part of any dependency chain. All dependency chains for which this is the case are added to a set, which is later returned to the calling function. Once the BFS has finished and *checkCallForDeps()* has returned the set of all instructions whose dependency chains either begin in or run through the called function, it initialises new dependency chains with the call's value or add the call's value to existing dependency chains respectively.

5 Implementation

5.1 Infrastructure

The passes for annotation and verification of address dependencies are implemented in a single source file as part of the LLVM source tree.

5.2 Data Types

We introduce several custom data types for annotation and verification of dependencies.

5.2.1 *Potential_Dependency_Half* Objects

For representing potential beginnings of dependencies, we introduce a *Potential_Dependency_Half* structure. Most importantly, it contains the current dependency chain for a given instruction and amongst others also a *call_stack*, implemented as a vector, for handling handling post dominance in interprocedural analysis correctly. Instructions map to *Potential_Dependency_Half* objects in DepChainMaps.

5.2.2 *Dependency_Half* Objects

As mentioned in 4, both passes maintain a map for beginnings and endings, each associating unique IDs with dependency beginnings / endings respectively. In source code, beginnings and endings are represented as *Dependency_Half* objects which store all required information for either making an annotation or printing a broken dependency, depending on the context. IDs are strings constructed from the address of the beginning instruction and the address of the ending instruction.

5.2.3 *BFS_Context* Objects

Furthermore, the BFS function, *checkBBForDeps()* and *checkCallForDeps()* all handle *BFS_Context* structures, storing what needs to be accessed or modified when traversing the CFG. Most notably, this includes the DepChainMap, a flag for checking whether the BFS is meant to annotate or verify, pointers to the beginning and ending maps and the current depth of recursion for interprocedural analysis.

5.3 The *InstVisitor* Pattern

As it is common for LLVM passes to iterate over a list of instructions and then take different actions based on the individual instruction kind, LLVM offers the *InstVisitor* design pattern. Note that Aside from convenience and readability, it does not provide any notable performance improvements over having a switch statement on an instruction's type [30]. Our *BFS_Context* structure inherits from LLVM's *InstVisitor* class and implements visitor functions for the following cases:

- The general *Instruction* case
- The *LoadInst* case
- The *CallInst* case
- The *ReturnInst* case
- The *StoreInst* case

This allows *checkBBForDeps()* to be implemented in 2 lines, as LLVM also provides a function for automatically visiting all instructions in a BB.

5.3.1 The General *Instruction* Case

For every dependency chain, the function checks whether any of the instruction's operand values are part of the dependency chain. If yes, the values of the instructions are added to the dependency chain.

5.3.2 The *LoadInst* and *StoreInst* Cases

The *LoadInst* and *StoreInst* cases are the most critical of all, as they mark ending, or in the case of a load also the beginning, of a dependency. Since loads and stores are handled similarly, both visitor functions forward to the *handle_load_store()* function.

5.3.3 The *handle_load_store()* Function

handle_load_store() first checks whether the memory access is marked *'volatile'*. If this is not the case, the pass simply checks if the memory access belongs to any dependency chain. For both, loads and stores, it checks whether the first operand, the source, is part of any existing dependency chain, and if this is the case, add the current memory access's value to said dependency chain. If the access is marked *'volatile'*, the visitor function switches between the cases of annotating and verifying.

Annotation Case

When annotating, the visitor function checks if the memory access is part of any dependency chain. For doing so, in the case of a load, it looks at its first and only operand, and in the case of a store, it looks at its second operand, the destination. Should the access be part of a dependency chain, *handle_load_store()* calls *addMetadataToPair()*. After checking existing dependency chains, in the case of a load, *handle_load_store()* also creates a new dependency chain as the volatile load marks the potential beginning of a new dependency.

Verification Case

When verifying, *handle_load_store()* hands off verification to *handle_annotation()* if *!annotation* type metadata is present.

5.3.4 The *CallInst* Case

After checking that the called function is not empty and that it is not variadic - this case is omitted - the *CallInst* visitor function builds a new context for the BFS to be started in the called function. One important difference is that the BFS in the called function does not receive the full dependency chains, but dependency chains which have been initialised with the relevant function argument. The dependency chains are only added if *will_always_reach(BB1, BB2)* evaluates to true, where *BB1* is the BB of the instruction marking the potential beginning and *BB2* is the current BB. Again, this is to rule control dependencies from being falsely identified as address dependencies and is further discussed in 6. The *CallInst* visitor function then calls *checkCallForDeps()*, and once it has received the return set, handles the dependencies which run through the called function or start in the called function. It is also here where the call is added to the call stack vector of the *Potential_Dependency_Beginning*.

5.3.5 The *ReturnInst* Case

If it is not called as part of interprocedural analysis or the return value is *void*, the *ReturnInst* visitor function immediately returns. Otherwise, the *ReturnInst* visitor function erases all dependency chains of instructions where the return value is not a part of their dependency chain.

$$"Doitlk: type, function\ name, line, call\ stack, id;"$$

Figure 5.1: The metadata strings our passes use for annotating and verifying dependencies in IR

5.3.6 Explicitly Skipped Cases

Since not all instructions are relevant for tracking dependency chains, e.g. because they do not 'return' a value, several instructions types are explicitly skipped by implementing an empty visitor function. These instruction types are:

- *AllocaInst*
- *CmpInst*
- *FenceInst*
- *AtomicCmpXchgInst*
- *AtomicRMWInst*
- *FuncletPadInst*

5.4 Annotation With and Verification of Metadata

During optimisations, IR code can be heavily modified. For being able to track 'interesting' instructions through the optimisation pipelines nevertheless, LLVM provides metadata, which can be attached to instructions or functions for example. Out of the box, LLVM provides the *!annotation* metadata type, allowing users to attach strings to instructions, which are guaranteed to be preserved through optimisations. Therefore, we are guaranteed to find all *!annotation* type metadata annotations again after optimisations if their corresponding instructions were not optimised away. *!annotation* type metadata was designed with the specific purpose of annotating 'interesting' instructions and therefore ideally suit our purpose [31]. We will see in 6 that *!annotation* type metadata is lost despite the guarantee of being preserved.

5.4.1 Representing Dependency Annotations as Strings

Since *!annotation* type metadata only supports strings, we define a string representation of the *Dependency_Half* objects as follows:

'DoitLK: ' specifies whether this is an annotation related to dependency orderings. It helps to differentiate our dependency annotations from other annotations LLVM might make.

'type' specifies whether this is a beginning or ending annotation.

'function name' is the name of the source code function the instruction belongs to.

'line' denotes the corresponding line number in source code. It is obtained through the *DebugLoc* object which can be attached to an instruction. If options rule out *DebugLoc*, the line number is set to *-1*.

'call stack' shows the call stack of functions through which the pass arrived at the instruction.

'ID' is used to uniquely identify a dependency pair.

!annotation type metadata is represented through a *MDNode* object in source code. An *MDNode* object may have several *MDOperand* objects attached which in the case of *!annotation* represent the different strings that were annotated to the instruction.

5.4.2 The *addMetadataToPair()* and *add_annotations()* Functions

There are two functions which deal with *!annotation* type metadata when running the annotation pass. *addMetadataToPair()* gets called every time the BFS identifies an address dependency in the *handle_load_store()* function. It simply adds the *Dependency_Half* objects to the *beginnings* and *endings* maps. *add_annotations* gets called once the BFS has finished and adds the right strings as *!annotation* type metadata to the instructions for every *Dependency_Half* object in *beginnings* and *endings*.

5.4.3 Verifying Dependency Annotations With *handle_annotations()*

handle_annotations() gets called every time the verification pass's BFS encounters *!annotation* type metadata in the *handle_load_store()* function. It looks at all *MDOperand* nodes attached to the *!annotation*, skipping those which do not contain the *'DoitLK'* identifier, and parses their string data through a helper function. If the ID of the dependency has been verified before, the current operand is skipped. This can happen when the same dependency is inlined in different contexts. In that case, it is encountered several times in the optimised IR although it originally only existed once. The IDs ensure that every annotation pair maps to exactly one dependency in the unoptimised IR and is only verified once. If the ID has not been verified before, *handle_annotations()* checks the annotation type. If it is a beginning, the *Dependency_Half* is inserted into the *beginnings* map and the dependency chain for the instruction is created for it to be tracked by the BFS. If it is an ending, *handle_annotations* attempts to verify the dependency by checking if the memory access is part of the correct dependency chain. *handle_annotations* continues with the next operand, leaving the broken dependency to be printed out at the end, if one of the following is true:

- A dependency chain with a matching ID does not exist.
- The correct dependency chain exists, but the current access does not depend on any value in it.
- The correct dependency chain exists, but the second access is not always reached by the first access, i.e. it is conditional.

If *handle_annotations* is able to verify the dependency however, all entries with the corresponding ID in *beginnings* and *endings* are deleted and the ID of the dependency is added to the set of verified IDs.

5.5 Breaking Dependencies for Testing - The *insertBug()* Function

The *insertBug()* function gets called conditionally at the end of a BFS in the annotation pass if the function's name corresponds to one of those in the testing module, i.e. its name contains the string *'doitlk'*. *insertBug()* artificially breaks either the beginning or the ending of the dependency chain in a given function. For both cases, the function iterates over the BBs in the function in arbitrary order and then over the instructions in each BB in order, looking for an instruction that contains a *!annotation* type metadata with the correct type annotation, i.e. beginning or ending. If it is able to find such an instruction, it attempts to break the beginning or ending of the corresponding dependency chain. In the case of it breaking a beginning, it inserts a bug value virtual register - *bugVal* in short - and initialise it with *'42'*. Starting at the beginning of a dependency, it continues looking for the first store instruction which uses the value obtained by the dependency's beginning and replace its first operand, i.e. the source, with the *bugVal* register, thereby breaking the dependency chain. Both, beginning and ending annotations, remain. In the case of it breaking the end annotation, again a *bugVal* is inserted and the source of the second load or store instruction is set to *bugVal*, thereby breaking the dependency chain, but once again preserving the annotations.

5.6 Determining Reachability

Every time the the BFS requires a check for post domination, it calls the *will_always_reach()* function, which accepts two instruction pointers and the call stack pointer for the first instruction as arguments. Unlike the built-in *dominates()* function for post-dominator trees in LLVM, *will_always_reach()* is able to handle instructions in different functions through the call stack it receives as a function argument. *will_always_reach()* returns true if every path leading to the end of the second instruction's function, starting from the first instruction or the call leading to the first instruction, reaches the second instruction.

5.7 Printing Broken Dependencies

print_broken_deps() finally prints all the dependencies which remain as potentially broken dependencies to the user. 5.2 shows how the function outputs a broken dependency to users. Due to the way LLVM handles printing of metadata, the *!annotation* metadata number in the instruction print does not match the *!annotation* metadata print below. We are not aware of a way of circumventing the mismatch as of now. Furthermore, the arrow at the end of *'via foo::50->'* is due to our current implementation and will be addressed in future version of our passes.

5.8 Clang Integration

In order for the passes to be run as part of the different Clang pipelines, they are injected using LLVM's callback interface [32]. The annotation pass is injected via the *registerPipelineStartEPCallback()* function and the verification pass is injected via *registerOptimizerLastEPCallback()*. If the Clang driver is invoked with the *'-O0'* option, the annotation pass is inserted via *registerOptimizerLastEPCallback()* and verification is skipped.

5.9 Running the Passes

Since the passes were directly inserted into the Clang optimisation pipeline, there is no difference between running a regular Clang build and our custom Clang build. We dive into more detail on building the Linux kernel with Clang in 6.

```
Address dependency with ID "0xc0768c00xc52d7e0" couldn't be verified. It
    ↪ might have been broken.

...First access in optimised IR
in function foo: %1 = load volatile i64, i64* %state, align 8, !
    ↪ annotation !12
!152 = !{!"DoitLK: address dep begin,foo,42,,0xc0768c00xc52d7e0;"}

Inst was originally found in function foo in line (source code) 42

<print IR of optimised foo function if available>

...Second access in optimised IR in function bar: %14 = load volatile i64,
    ↪  i64* %state.i, align 8, !annotation !16
!35 = !{!"DoitLK: address dep end,bar,21,foo::50-> ,0xc0768c00xc52d7e0;"}
Inst was originally found in function bar in line (source code) 21
via foo::50->

<print IR of optimised bar function if available>
```

Figure 5.2: Printing a broken address dependency

6 Evaluation

With the uncertainty of dependency orderings being broken, there is a chance of our tool functioning correctly, yet it finding no instances of broken dependencies in the kernel. For testing our implementation, we construct relevant dependencies in a Linux kernel module which are either artificially broken or explicitly not detected by our implementation, e.g. because they are control dependencies. We base our test cases on [33] and [34] which discuss *memory_order_consume* dependency chains in C.

6.1 Experimental Setup

Since we have implemented annotation and verification of dependencies as well as the insertion of artificially broken dependencies as LLVM compiler passes, which have directly been integrated into Clang, there are no additional changes required to build a Linux kernel with our passes enabled. We maintain an LLVM source tree which includes our passes, as well as a Linux kernel source tree, where the kernel module with the relevant test cases has been added in a new *lib/modules* directory and integrated into the KBUILD build system. Our upper bounds for interprocedural analysis are set to '3' for annotation and '4' for verification. We only build for arm64 as it is a weakly-ordered architecture and therefore relevant for broken dependency orderings. If all dependencies are present, the commands in 6.1 build a Linux kernel for arm64 based on *allyesconfig*. Our testing was done with the most recent version of the Linux kernel - at the time of writing v5.15 rc7 - and LLVM 13. We build the Linux kernel with a slightly modified version of *allyesconfig* where sanitisers have been disabled to improve readability of IR code. In the following, we refer to it as *doitlkconfig*. We run our builds with NixOS version '21.05.20211022.1762637' on an AMD EPYC 7713P x86 CPU with 512GB of RAM.

6.2 Overheads

We identify the overhead the passes introduce by comparing *doitlkconfig* Linux kernel builds with and without our passes enabled. We use the command shown in 6.2. Results are shown in 6.3. The passes introduce an overhead of roughly 10 seconds. This

```
make allyesconfig
```

```
make CC=<path\_to\_LLVM>/build/bin/clang ARCH=arm64 CROSS\_COMPILE=
    ↪ aarch64-unknown-linux-gnu- modules\_prepare
```

```
make CC=<path\_to\_LLVM>/build/bin/clang ARCH=arm64 CROSS\_COMPILE=
    ↪ aarch64-unknown-linux-gnu-
```

Figure 6.1: Commands for compiling a Linux kernel with our passes enabled

```
time (make -s CC=<path_to_LLVM>/build/bin/clang ARCH=arm64 CROSS_COMPILE=
    ↪ aarch64-unknown-linux-gnu- -j128 -s)
```

Figure 6.2: Command for timing a Linux kernel build

does not even make up one percent of the build time without passes enabled, and we therefore deem it appropriate.

6.3 Current Restrictions

Generally, we favour false negatives over false positives for building trust in the tool. As a result of not being able to handle control dependencies yet, we require dependency chains to be strictly unconditional, which is tested by the AD7 case. This means that no part of a dependency chain can be control-flow dependent or if it is, then it must appear in all branches control flow might take. Furthermore, we rule out functions with variadic argument lists and function pointers. Also, we only detect address dependencies where both accesses are marked as per LKMM.

	Output
With passes enabled	43043.29s user 2747.37s system 9642% cpu 7:54.90 total
With passes disabled	42483.07s user 2758.21s system 9726% cpu 7:45.12 total

Figure 6.3: A comparison of build times when building the Linux kernel with *doitlkconfig* and our passes either enabled or disabled

6.4 Testing Our Implementation

We define the following dependencies as test cases, where each dependency gives way for two test cases, breaking the beginning and breaking the ending of the dependency. If not specified otherwise, each test cases is implemented as a read -> read as well as a read -> write address dependency. Control dependencies are generally only implemented for the read -> write case since LKMM does not guarantee ordering for read -> read control dependencies [5]. We abbreviate address dependencies with AD and control dependencies with CD and simplify function names. When possible, we stick to the naming scheme of [33]

6.4.1 Address Dependencies

AD1: Simple Case

This is the trivial test case. The dependency begins and ends in the same function.

AD2: In via Function Parameter

Our second dependency starts in function A, then runs into function B through a dependent argument in a function call and ends in function B.

AD3: Out via Function Return

AD3 sees function A calling function B with no arguments. An address dependency then begins in function B, where function B's return value is part of the dependency chain. The dependency then ends in function A after being returned from function B.

AD4: In and Out, Same Chain

The dependency begins in function, runs through into function B through a function call, gets returned and finally ends in function A.

AD5: Simple Case - End in If Condition

AD5 matches *AD1* apart from the dependency ending in an if condition here. This case only applies to read -> read dependencies as *WRITE_ONCE()* cannot be converted to a Boolean value and can therefore not be used in an if condition.

AD6: Simple Case - Chain Through If-Else

AD6 tests the case where a part of the dependency chain is conditional, but it appears in all the branches control flow could take, allowing our passes to consider it nevertheless.

AD7: Simple Case - Chain Through If

AD7 marks a case which is explicitly ruled out. A part of the dependency chain is conditional and therefore not considered by our passes. This case marks a control dependency as well as an address dependency. When the if-condition evaluates to true, the address dependency gets enabled. However, for now, this case is not annotated by our passes.

AD8: Simple Case - Fanning Out

AD8 tests two address dependencies with the same beginning, within the same function, but different endings.

AD9: Fanning Out

AD9 contains two dependencies with the same beginning. The dependencies start at the same load in function A and carry on with the same dependency chain until they fan out into functions B and C where both dependencies end.

AD10: In and Out, but Different Chains

AD10 can be either understood as a unification of *AD2* and *AD3* or a variation of *AD4*. An address dependency starts in function A and ends in function B, whilst another dependency starts in function B and ends in function A.

Isolated *__ktime_get_fast_ns()* Case

We have isolated the address dependency from *kernel/time/timekeeping.c::__ktime_get_fast_ns()* which was discussed in [4]. We edit the code such that the second load is also annotated as shown in 6.4 since our passes do not yet cover the case where the second load is not annotated. We do not break this dependency for testing and since it was taken from Linux kernel source code, we only consider the read -> read case.

```
seq = raw_read_seqcount_latch(&tkf->seq);
tkr = tkf->base + (seq & 0x01);
now = ktime_to_ns(READ_ONCE(tkr->base));
```

Figure 6.4: A modified version of the address dependency discussed in [4]

6.4.2 Control Dependencies

We add several control dependency test cases which are not annotated by the passes. All CD test cases occur within the same function.

CD1: Simple Case - If Condition Not Part of Dependency Chain

This marks a trivial control dependency within the same function where the if condition is not part of the dependency chain.

CD2: Simple Case - If Condition Part of Dependency Chain

CD2 is identical to *CD1* except that this time the if condition is part of the dependency chain.

CD3: Simple Case - If Branch Cannot Be Reached

CD3 is again identical to *CD1*, but the if condition is such that it always evaluates to false.

CD4: Simple Case - Beginning in If Condition

The *CD4* dependency begins in the if condition - an assignment is implicitly converted to a Boolean value - and ends in the if branch.

CD5: Simple Case - Ending in For Loop

Since for loops are executed conditionally as well, we consider them as control dependencies. *CD5* differs from *CD1* in the sense that it has a for loop instead of an if condition.

6.5 Findings

6.5.1 Annotated Versus Verified Dependencies

Excluding test cases and broken dependencies, with *doitlkconfig*, our passes annotate 1197 and verify 331 address dependencies. We attribute the discrepancy between the two numbers to the passes only verifying a given ID once although it may have been annotated multiple times with different call stacks. Also, dependencies might be optimised away.

6.5.2 Dependencies Flagged as Potentially Broken

Lost Annotations

Our passes flag two cases in *drivers/md.c::state_show()* and *net/core/dev.c::napi_enable()*, both ending in *test_bit()*, as broken where the dependency in fact got preserved, but the *!annotation* type metadata got lost. This, by design, should not happen, and we plan to investigate this further by reaching out to the appropriate mailing lists.

Finding a Dependency Which Is Syntactic, but Not Semantic

We found a potentially broken address dependency in *fs/afs/addr_list.c::afs_iterate_addresses()*, lines 375 - 377. 6.5 shows the relevant dependency which we took to the Linux kernel mailing list [35]. As it turns out, it is expected that the compiler can break this dependency as it is only syntactic, not semantic. It is syntactic in the sense that there is an array subscript operator in *addr[BIT_WORD(nr)*, computing an address by using a value which traces back to a *READ_ONCE()*. However, it is not semantic as the index which is being computed inside the array subscript operator must always evaluate to zero, as *addr* points to an *unsigned long*. As a result of the discussion on LKML, we will submit a pull request for the LKMM documentation, explicitly outlining the ambiguity in syntactic and semantic address dependencies.

6.6 Implementation Improvements

With the eventual goal of upstreaming the tool once it is reliably able to identify address and control dependencies, there are several implementation improvements that we want to make. Primarily this concerns a tighter integration with existing LLVM code, e.g. by leveraging existing passes if possible or LLVM-specific data types such as the *SmallVector<>* or *SmallString<>* templates. Specifically, we want to explore an integration with the *MemoryDependenceAnalysisPass* [36] and determine whether def-use

```
[...]
index = READ_ONCE(ac->alist->preferred);
if (test_bit(index, &set))
    goto selected;
[...]
```

```
arch_test_bit(unsigned int nr, const volatile unsigned long *addr)
{
    return 1UL & (addr[BIT_WORD(nr)] >> (nr & (BITS_PER_LONG-1)));
}
```

Figure 6.5: An address dependency which is purely syntactic (and not semantic) in *fs/afs/addr_list.c*

/ use-def chains would be suitable for tracking dependency chains. We hat initially investigated def-use / use-def chains, but abandoned the approach as a result of heavy changes in our implementation. Finally, we would like to iron out small issues with the printing of broken dependencies, e.g. make metadata numbers match and remove the redundant arrow at the end of call stacks as mentioned in 5.

7 Related Work

7.1 Addressing Consume Semantics on the Language Level

On a C language level, there exist several approaches for enabling compilers to track *memory_order_consume* dependencies [37] [38]. Consume semantics are directly related to our problem as they are concerned with tracking the same kinds of dependency chains. Whether these approaches are of use to the Linux kernel we will only be able to tell once a definitive version make is into the C standard.

7.2 Linux Kernel-Specific Approaches

On a Linux-kernel level, there have been proposals for marking control dependencies in code, specifying the programmers intent, thereby allowing compilers to preserve the dependencies. Such approaches have been met with criticism, most notably by Linus Torvalds, as the significance of the problem is still uncertain and it yet has to be empirically shown that such annotation are indeed required, which we attempt to address with our work [39] [40]. We are not aware of any approaches using compiler passes for identifying potentially broken dependency orderings in the Linux kernel.

7.3 Rust in the Linux Kernel

As of April 21, using Rust in the Linux kernel is being heavily discussed [41]. At the moment, Rust uses the C11 memory model and the jury is still out as to how Rust will work with LKMM. [42]

7.4 XNU-Darwin Kernel

The XNU-darwin kernel, which is used for Apple's macOS and iOS operating systems, discourages the use of *memory_order_consume* in its documentation and instead proposes the use of its custom *dependency* memory order. Like Linux, XNU relies on its own implementation of atomics and to compensate for the current state of *memory_order_consume*, it has implemented the *dependency* memory order. However, it caveats its use by pointing out that compilers might still be able to break such dependencies if they are able to infer certain properties about the pointers being used, e.g. deduce that they can only be from a set of a few constants for example [43]. This seems to point to the discrepancy between syntactic and semantic address dependency which we discussed in 6. It appears that XNU's implementation of the *dependency* order relies on inline assembly, making it resistant against harmful compiler optimisations [44]. Yet, it sparks our interest and we would like to look into it further, hoping that it would mark a step in generalising our tool.

8 Future Work

The immediate next step we want to take with the project is expanding our tool such that it is able to check for broken control dependencies. We believe that with the existing algorithm and checks for post-domination in place, achieving control dependency support is not far off. Furthermore, we would like to have our passes run with LTO, and we want to expand our search for broken dependencies by looking at other weakly-ordered architectures and more flavours of the Linux kernel, e.g. Android, which is built with Clang by default and often with LTO enabled [45], too. When exploring other flavours of Linux, we would use the opportunity to further investigate how and if our problem relates to the XNU kernel. We would like to eliminate the two false positives we are currently seeing and want to reach out to the appropriate mailing lists to identify why the annotations are being lost. We aim to extend our implementation such that it can run in *strict* and *relaxed* identification modes, where the latter would not require the second memory access to be marked *'volatile'*, i.e. it does not pose a data race. This would significantly increase the amount of tracked dependencies, but also the risk of false positives. Once we have an implementation which is reliably able to identify broken address and control dependencies, we want to look at upstreaming the passes into either the Linux kernel or LLVM source trees. One step which must need to be taken before is to tighten the integration of our existing implementation with LLVM, be it through the use of LLVM-specific data types or by building on existing compiler passes as discussed in 6. And finally, we want to get our work published - together with concrete instances of relevant dependencies being broken we hope.

9 Conclusion

Whilst we were not yet able to find the 'holy grail' of a truly broken dependency ordering in the Linux kernel, we believe that we have built a solid foundation for reliably identifying broken address and control dependencies. We were able to implement an efficient solution for tracking a subset of address dependencies which integrates with the Clang compiler and is able to run as part of the Linux kernel build process. We were already able to contribute to LKML, be it an edge case, and are in the process of submitting a pull request for the LKMM documentation. We have identified immediate TODOs in 8, and since we now have the relevant infrastructure ready, we aim for significant progress in the coming months.

List of Figures

List of Algorithms

Bibliography

[1] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA: Association for Computing Machinery, Mar. 2018, pp. 405–418, ISBN: 978-1-4503-4911-6. [Online]. Available: https://doi.org/10.1145/3173162.3177156 (visited on 11/13/2021).

[2] L. Torvalds, *Re: [RFC] LKMM: Add volatile_if()*, Sep. 2021. [Online]. Available: https://lore.kernel.org/lkml/CAHk-=wgJZVjdZY07iNb0hFz-iynrEBcxNcT8_u317J0-nzv59w@mail.gmail.com/ (visited on 08/15/2021).

[3] P. E. McKenney, U. Weigand, A. Parri, and B. Feng, *Linux-Kernel Memory Model*, Sep. 2017. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0124r4.html (visited on 08/15/2021).

[4] W. Deacon, *Dependency Ordering in the Linux Kernel*, 2020. [Online]. Available: https://linuxplumbersconf.org/event/7/contributions/821/attachments/598/1075/LPC_2020_--_Dependency_ordering.pdf.

[5] D. Howells, P. E. McKenney, W. Deacon, and P. Zijlstra, *Linux Kernel Memory barriers*. [Online]. Available: https://www.kernel.org/doc/Documentation/memory-barriers.txt (visited on 04/20/2021).

[6] W. Deacon, *The Never-Ending Saga Of... Control Dependencies*, 2021. [Online]. Available: https://linuxplumbersconf.org/event/11/contributions/973/attachments/810/1587/Control%20deps%20-%20LPC%202021.pdf.

[7] N. Desaulniers, *Re: [PATCH v2 18/18] arm64: Select ARCH_SUPPORTS_LTO_CLANG*. [Online]. Available: https://lore.kernel.org/linux-arm-kernel/CAKwvOdkDz1yRo=Skt_mWxsOuucK+adXx8kFQksMbTE3ofnpUMQ@mail.gmail.com/ (visited on 09/23/2021).

[8] W. Deacon, *Re: [PATCH 00/22] add support for Clang LTO*. [Online]. Available: https://lore.kernel.org/linux-arch/20200701100358.GA14959@willie-the-truck/ (visited on 09/23/2021).

[9] M. Elver, *D103958 [WIP] Support MustControl conditional control attribute*. [Online]. Available: https://reviews.llvm.org/D103958 (visited on 09/26/2021).

[10] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996, ISSN: 1558-0814. DOI: `10.1109/2.546611`.

[11] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979, ISSN: 1557-9956. DOI: `10.1109/TC.1979.1675439`.

[12] L. Maranget, S. Sarkar, and P. Sewell, "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models," [Online]. Available: `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`.

[13] *Memory_order - cppreference.com*. [Online]. Available: `https://en.cppreference.com/w/c/atomic/memory_order` (visited on 09/22/2021).

[14] *Intel 64 Architecture Memory Reordering Whitepaper*, Aug. 2007. [Online]. Available: `https://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf` (visited on 09/20/2021).

[15] "The SPARC Architecture Manual Version 8," p. 295,

[16] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: X86-TSO," in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407, ISBN: 978-3-642-03358-2. DOI: `10.1007/978-3-642-03359-9_27`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-03359-9_27` (visited on 09/20/2021).

[17] *ARMv8-A Memory systems*. [Online]. Available: `https://developer.arm.com/documentation/100941/0100/The-memory-model?lang=en` (visited on 10/04/2021).

[18] *LLVM Users*. [Online]. Available: `https://llvm.org/Users.html` (visited on 10/03/2021).

[19] *The LLVM Compiler Infrastructure Project*. [Online]. Available: `https://llvm.org/` (visited on 10/03/2021).

[20] *Clang Description*. [Online]. Available: `https://clang.llvm.org/docs/CommandGuide/clang.html` (visited on 11/13/2021).

[21] *Clang Code Generation Options*. [Online]. Available: `https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options` (visited on 11/14/2021).

[22] *LLVM FAQ*. [Online]. Available: `https://llvm.org/docs/FAQ.html#can-i-compile-c-or-c-code-to-platform-independent-llvm-bitcode` (visited on 10/03/2021).

[23] D. Howells, *Re: [RFC][PATCH 0/5] arch: Atomic rework*. [Online]. Available: `https://yhbt.net/lore/all/21984.1391711149@warthog.procyon.org.uk/` (visited on 09/23/2021).

[24] *Re: [PATCH 00/22] add support for Clang LTO - Marco Elver*. [Online]. Available: `https://lore.kernel.org/linux-arch/20200630191931.GA884155@elver.google.com/` (visited on 10/16/2021).

[25] *Building Linux with Clang/LLVM*. [Online]. Available: `https://www.kernel.org/doc/html/latest/kbuild/llvm.html` (visited on 09/23/2021).

[26] *LLVM Language Reference Manual - Volatile*. [Online]. Available: `https://llvm.org/docs/LangRef.html#volatile-memory-accesses` (visited on 09/26/2021).

[27] *Rwonce.h*, Sep. 2021. [Online]. Available: `https://github.com/torvalds/linux/blob/58e2cf5d794616b84f591d4d1276c8953278ce24/include/asm-generic/rwonce.h` (visited on 09/23/2021).

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, ISBN: 978-0-262-03384-8.

[29] *LLVM PostDominatorTree Class Reference*. [Online]. Available: `https://llvm.org/doxygen/classllvm_1_1PostDominatorTree.html` (visited on 10/30/2021).

[30] *InstVisitor.h*, LLVM, Oct. 2021. [Online]. Available: `https://github.com/llvm/llvm-project/blob/d054b80bd3ab1a78d1a870f941024429273d2a83/llvm/include/llvm/IR/InstVisitor.h#L33` (visited on 10/25/2021).

[31] F. Hahn, *D91188 Add !annotation metadata and remarks pass.* [Online]. Available: `https://reviews.llvm.org/D91188` (visited on 09/13/2021).

[32] *LLVM PassBuilder Class Reference*. [Online]. Available: `https://llvm.org/doxygen/classllvm_1_1PassBuilder.html` (visited on 11/13/2021).

[33] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, N. Clark, O. Giroux, L. Crowl, J. Bastien, and M. Wong, *Marking memory order consume Dependency Chains*, 2017. [Online]. Available: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0462r1.pdf` (visited on 09/21/2021).

[34] P. E. McKenney, T. Riegel, J. Preshing, H. Boehm, C. Nelson, O. Giroux, and L. Crowl, *P0098R0: Towards Implementation and Use of Memory Order Consume*, 2015. [Online]. Available: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0098r0.pdf`.

[35] A. Stern, *Re: Potentially Broken Address Dependency via test_bit() When Compiling With Clang*. [Online]. Available: `https://lore.kernel.org/all/20211028143446.GA1351384@rowland.harvard.edu/` (visited on 11/06/2021).

[36] *LLVM: MemoryDependenceAnalysis Class Reference*. [Online]. Available: https://llvm.org/doxygen/classllvm_1_1MemoryDependenceAnalysis.html (visited on 10/04/2021).

[37] J. Bastien and P. E. McKenney, *P0750r1: Consume*, Nov. 2018. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0750r1.html (visited on 11/08/2021).

[38] P. E. McKenney, M. Wong, H. Boehm, J. Maurer, J. Yasskin, and J. Bastien, *Proposal for New memory order consume Definition*, Feb. 2017. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0190r4.pdf (visited on 09/22/2021).

[39] P. Zijlstra, *[RFC] LKMM: Add volatile_if()*, Apr. 2021. [Online]. Available: https://lore.kernel.org/lkml/YLn8dzbNwvqrqqp5@hirez.programming.kicks-ass.net/ (visited on 11/08/2021).

[40] M. Desnoyers, *[RFC PATCH] LKMM: Add ctrl_dep() macro for control dependency*, Sep. 2021. [Online]. Available: https://lore.kernel.org/all/20210928211507.20335-1-mathieu.desnoyers@efficios.com/T/#m782fa8f16d5763b54daa87b73372ab9df0883b25 (visited on 11/08/2021).

[41] M. Ojeda, *[RFC] Rust support*, Apr. 2021. [Online]. Available: https://lore.kernel.org/lkml/20210414184604.23473-1-ojeda@kernel.org/ (visited on 11/08/2021).

[42] P. E. McKenney, *So You Want to Rust the Linux Kernel?* Sep. 2021. [Online]. Available: https://paulmck.livejournal.com/62436.html (visited on 10/09/2021).

[43] *XNU use of Atomics and Memory Barriers*, Apple, Nov. 2021. [Online]. Available: https://github.com/apple/darwin-xnu/blob/2ff845c2e033bd0ff64b5b6aa6063a1f8f65aa32/doc/atomics.md (visited on 11/08/2021).

[44] *XNU atomic_private_arch.h*, Apple, Nov. 2021. [Online]. Available: https://github.com/apple/darwin-xnu/blob/2ff845c2e033bd0ff64b5b6aa6063a1f8f65aa32/libkern/os/atomic_private_arch.h (visited on 11/08/2021).

[45] *Android - Building Kernels*. [Online]. Available: https://source.android.com/setup/build/building-kernels (visited on 11/08/2021).