SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Out-Of-Spec Compilation in the Presence of Dragons: Investigating Broken Dependency Orderings in the Linux Kernel

Paul Heidekrüger

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Out-Of-Spec Compilation in the Presence of Dragons: Investigating Broken Dependency Orderings in the Linux Kernel

Spezifikationsabweichendes Kompilieren in der Gegenwart von Drachen: eine Untersuchung verletzter Abhängigkeitsordnungen im Linux Kernel

Author:	Paul Heidekrüger
Supervisor:	Prof. Pramod Bhatotia
Advisor:	Dr. Marco Elver (Google)
Submission Date:	November 7, 2023

I confirm that this master's thesis is my own work, and I have documented all sources and material used.

Munich, November 7, 2023 Paul Heidekrüger

Acknowledgments

I would like to thank my main advisor, Marco (Elver), for his expertise and support throughout my work since my bachelor's thesis. Our regular meetings and Marco's advice have proven invaluable for my progress. Marco initially suggested this work as a research project, probably not knowing that it would lay the foundation for what will hopefully become a meaningful career in research and beyond for myself. Further, I would like to thank Pramod (Bhatotia) and Soham (Chakraborty) for their advice and for supporting my work since my bachelor's thesis. Without Marco, Pramod, and Soham, this work would not have been possible, and I'm beyond grateful not only for having been given the opportunity to work on this project in the first place but for being able to continue this work after my bachelor's thesis and hopefully well after this master's thesis with more projects to come. In addition, I would like to thank Martin Fink for his work on implementing an address dependency checker in the LLVM backend.

Thank-yous must also be given to Paul E. McKenney, Will Deacon, and the ClangBuiltLinux community (especially Nick Desaulniers) for their interest in this work and for providing much-needed feedback and input along the way.

Finally and above all, I want to thank my parents and grandparents for their continuous encouragement and support during my studies.

Contents

Part I.

Preface

A Note on Previous Work

This master's thesis is based on the following previous work:

- My bachelor's thesis "Dependency Ordering in the Linux Kernel" (submitted in November 2021).
- A guided research conducted in the following semester and concluded in April 2022.
- Work done between the guided research and this master's thesis, which was not part of any university course.

Abstract

A unique discrepancy between how the Linux kernel and modern compilers reason about concurrency in the C language has led to the Linux kernel being at risk of being miscompiled. In particular, the Linux kernel's dependency orderings are at stake, which, if broken due to a miscompilation, can lead to bugs in multi-threaded Linux kernel code, which keep themselves well hidden from users, as they will only be visible after compiler optimizations have run. This master's thesis continues our work on automatically identifying and reporting broken dependency orderings during kernel compilation. We improve our existing static dependency-checking mechanism and provide a proof-of-concept, which extends our static analysis through interleaved symbolic execution to check for broken dependency orderings at runtime.

1. Introduction

The quest for investigating broken dependency orderings in the Linux kernel continues. Broken dependency orderings in the Linux kernel pose a unique problem, and discussions have been heating up ever since the introduction of the C11 standard [How14]. At the core of the problem lies the fact that the Linux kernel deliberately operates "outside the spec" [Tor12], mainly for performance reasons, but in the case of atomics, simply due to them being required before they were offered by the C standard [How06]. Hence, the Linux kernel's atomic accesses come with their own memory model, the Linux-kernel memory model (LKMM).

The LKMM is one of the main arguments for considering the Linux kernel to be written in its own dialect of standard C. One might refer to this as Linux C. The LKMM defines several kinds of dependency orderings in Linux C. When multiple threads interact, the LKMM dependency orderings prevent certain dependent instructions from becoming visible to other threads out of order.

However, as these are defined in Linux C and not standard C, modern compilers are unaware of the LKMM's dependency orderings. Given the growing extent of compiler optimizations, this poses a problem. This problem manifests itself in compiler optimizations that break dependency orderings. That means that if one were to look at a reverse-engineered source code representation of the optimized code, one would not be able to infer the initial dependency ordering. On weakly-ordered CPU architectures, such as ARM or PowerPC, this becomes an issue, as it is now possible for the CPU to reorder the dependent so that another thread could observe them happening out of the intended program order. Programmers would be left wondering, as they usually do not see the optimized code that the compilers produce, and such behavior contradicts the LKMM.

With performance-critical code at stake, the Linux kernel community has grown increasingly concerned about such hard-to-debug miscompilations. Eventually, that concern led to two talks at the Linux Plumbers Conference [Dea20] [Dea21], which laid the foundation for this work. Our bachelor's thesis [Hei21] proposed a first version of an address dependency tracking mechanism, which still lacks evidence of real broken dependency orderings in the Linux kernel. Our work continued, leading to a talk of our own at the Linux Plumbers Conference, presenting evidence of broken dependency orderings in the Linux kernel [EH22], and it finally led to this master's thesis.

1.1. Contributions

This master's thesis presents ongoing work on broken dependency orderings in the Linux kernel and makes the following main contributions:

- In §6, we outline the StatDepChecker, an improved version of the static address dependency checking mechanism proposed in our bachelor's thesis [Hei21].
- In §7, we outline the DynDepChecker, a proof-of-concept runtime dependency checker which can use the annotations made by the static dependency checker.
- In §6.3, we present more evidence of broken dependency orderings when compiling a Linux kernel.

Part II.

Background

2. Program Optimization

Today's computer architectures and compiler toolchains enable significant performance improvements by performing program optimizations at compile time and runtime. The following will give a broad overview of both, starting with optimizing CPUs.

2.1. Optimizing CPUs

Before we discuss optimizing CPUs any further, we must define some key terms to avoid ambiguities.

Hardware thread A stream of instructions within a CPU core.

CPU core The hardware unit within a CPU that is able to execute instructions. A CPU core offers one or more hardware threads, each of which usess the same underlying hardware.

Uniprocessor system Multicore CPUs and simultaneous multithreading make the term "uniprocessor system" ambiguous. For our purposes, a uniprocessor system has a single CPU with a single core, offering one hardware thread.

Multiprocessor system Again, "multiprocessor system" is an ambiguous term, which we define as a system that has more than one hardware thread. That may be a single-core system with multiple hardware threads, a multicore system with one or more hardware threads per core, or a system with more than one CPU, each potentially having multiple cores with potentially multiple hardware threads per core.

Parallelism By Michael J. Scott's definitions [Sco13], we define two operations as being parallel if they "may [be executed] at the same time."

Concurrency The term "parallel" is not to be confused with the term "concurrent," which we define as "[two operations having] started but neither [having] finished." Parallelism can, therefore, be considered an "implementation" of concurrency [Sco13].

Let us consider a uniprocessor system. Even though programmers may perceive the execution of a program on a uniprocessor system to happen in the order specified in the source code, the execution of individual instructions happens in parallel. The term "core" is merely an abstraction for an array of functional units that implement the illusion of executing one instruction after another. One key innovation to enable instruction-level parallelism is called pipelining.

2.1.1. Pipelining

The key idea behind pipelining is not to increase the execution speed of a given instruction but to increase the overall throughput by working on several instructions in parallel [PH17]. A pipelined CPU core works under the assumption that every instruction in the instruction set architecture (ISA) requires the same number of steps to complete. The assumption also works in reverse, as CPU designers have these steps in mind when designing new architectures that support pipelining. These steps could be, for instance:

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

Each step is implemented in a specialized functional unit within the CPU core. That means that once the first instruction has completed the first step in the pipeline, the second instruction can already be fetched from memory. This continues for all the pipeline stages until all functional units are busy. In an ideal world, there would be no clock cycle where a functional unit is not busy, and instructions would transition between stages every clock cycle, yielding a theoretical speed up by the factor of the number of pipeline stages. There are several reasons why such a speed-up is not always possible. An instruction might need significantly more time per pipeline stage than others, and we did not account for the overhead required to set up and maintain the pipeline, for instance. Another major reason, and this is why the distribution of instructions amongst the pipeline stages is anything but trivial, are hazards.

Hazards

Hazards describe relations on instructions. A hazardous instruction pair will prevent the pipeline from progressing if left unaddressed. Hennessey and Patterson [PH17] describe three kinds of hazards: structural hazards, data hazards, and control hazards. A pipeline faces a **structural hazard** when a given combination of instructions cannot be executed due to a hardware limitation, e.g. because two different instructions require the same hardware resource in their respective stages, say, a memory or floating point unit. A pipeline faces a **data hazard** when one instruction cannot execute before another instruction completes because of a data dependency. A dependent instruction may be an addition that adds something to a value another instruction returns, or a dependent instruction may be a store that requires another instruction to return for being able to compute its destination address. A pipeline faces a **control hazard** when "the flow of instruction addresses is not what the pipeline expected." Or in more concrete terms, when it has not yet been determined whether a certain set of instructions can be executed. For example, the pipeline may expect a branch but is given a set of unrelated instructions because the corresponding branch condition has not been computed yet.

Maintaining the Illusion of Program Order

Hazards are a result of physical limitations. They exist because CPUs cannot predict the future. Hazards, therefore, mark the most basic kinds of dependency orderings CPUs must respect. If CPUs were not to respect hazards, they would be considered faulty, as they would compute incorrect results. However, hazards are only defined for a single stream of instructions. On multiprocessor systems, even as hardware thread maintains the "illusion" of program order, the behavior of one core affects the others. This will be further discussed in §3. Real pipelines are much more involved than the above explanation makes it out to be. Real CPUs will track the out-of-order execution of instructions, predict branch results, and identify loops. The above explanation's goal was merely given to illustrate hazards.

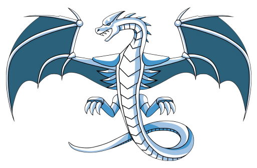
2.2. Optimizing Compilers

Modern software development relies on higher-level languages, with compilers providing the interface between high-level languages and the low-level machine code CPUs understand. Compilers comprise a multi-stage process, which, depending on configuration, heavily optimizes the program while progressing through increasingly concrete layers of abstraction until a binary is generated. The term "compiler", like the term "CPU", is an abstraction for, in this case, an array of programs. Traditionally, it only refers to a piece of software that is able to translate a program from a higher-level programming language, say C, into another language, say X86 assembly. Nowadays, that definition is often stretched and refers to the process of going from a higher-level programming language to an executed binary. Unlike the other definition, to adjust for modern optimizing compilers, this definition includes the driver for, i.e., the program users invoke to start a compilation, assembling, and linking, which are traditionally not considered to be a part of compiling a program [CT12].

The optimizations compilers can make are reigned in by the respective programming language as well as the specification of the architecture that is being targeted. In the case of C, there exists a meticulously drafted language standard, to which any C compiler must adhere. For instance, for accesses marked volatile, it stipulates that "[a]ctions on objects [declared volatile] shall not be 'optimized out' by an implementation or reordered except as permitted by the rules for evaluating expressions" [WG118].

2.3. LLVM

LLVM brands itself as a "collection of modular and reusable compiler and toolchain technologies" [LLVg], which includes the well-known and widely-used clang compiler. Although it began its life as a research project, LLVM is widely used in industry and academia. The LLVM logo shows a dragon [LLVc].



2.3.1. LLVM's Subprojects

Since the LLVM repository should be considered a collection and not a coherent piece of software, it requires some untangling. The name LLVM itself is ambiguous. It used to be an acronym for "low-level virtual machine," but as of today stands on its own. To the best of our knowledge, it currently has at least three meanings, depending on context.

1. It refers to the LLVM repository and therefore all projects that exist under the LLVM umbrella.

2. It refers to the optimizations being performed by some subproject in the LLVM repository.

3. It refers to the clang compiler.

When we use the term LLVM on its own, we intend it to refer to the LLVM source code repository. Any other meaning will be made explicit through context. LLVM's subprojects are recursive in the sense that some of LLVM's subprojects will be used to construct a piece of software, which then in turn becomes an LLVM subproject to be re-used in the future. One example of this is the clang compiler which uses LLVM to provide a production-ready compiler for the C language family [LLVf].

2.3.2. The LLVM Architecture

From the beginning, LLVM was designed to provide what is still one of its main selling points today: modularity [Lat02] [Lat11]. Modularity is enabled by directing subprojects towards the traditional architecture for optimizing compilers, consisting of a three-stage architecture. The three stages are the **frontend**, the **middle-end** (or optimizer), and the **backend**. Each of these stages may be implemented through one or several subprojects in LLVM, and inter and intra-stage communication and progression between stages happens through clearly defined interfaces.

LLVM Frontend

The frontend is responsible for parsing the source code. In the process, it will generate an abstract syntax tree (AST), allowing it to catch syntax errors and perform minor optimizations, and finally an intermediate representation (IR) of the code. By its nature, a frontend is source code-dependent.

LLVM Middle-End

The middle-end's main job consists of analyzing and optimizing the IR generated by the frontend. Analyses and optimizations are performed by LLVM compiler passes. Different passes may depend on each other and are scheduled for different units of IR, e.g. functions, loops, or modules. For example, an analysis pass can be used to build a data structure, e.g. dominator tree, which then helps an optimization pass make decisions on whether a function should be inlined or not [LLVe]. IR is organized into modules. Each module corresponds to one (or several if merged) translation units. Each module contains functions. Defined functions consist of a control flow graph made up of basic blocks. Each basic block consists of a number of IR instructions. Instructions have a value and operands.

LLVM Backend

The backend takes the optimized IR and generates the program in the target ISA's language. This process includes target-specific optimizations.

2.3.3. LLVM's Intermediate Representation

LLVM IR plays a crucial role in decoupling the different stages. While it is technically target-dependent [LLVa], it is generic enough for the same optimizations to be used across different programming languages and, potentially with some tweaks, can be handled by the target architecture's backend independently of what source language it came from. Technically, that makes clang a frontend; to compile a new source code language, one merely needs to implement the process of going from source language to IR. For the rest, existing LLVM subprojects can be reused. Of course, clang grew larger than the mere frontend it initially was, but it remains a prime example of how to use LLVM's subprojects to build a compiler without having to re-implement every step of the way.

3. Memory Consistency Models

In §2.1 we gave an example of how modern CPUs perform optimizations while still maintaining the illusion of program order to its users. Our discussion focussed on a single CPU core. However, today, uniprocessor systems are hardly common. Not only are multi-core processors a given in today's server landscape, but as of the last decade, they have become ever-present in the consumer market in personal computers and mobile devices, making an understanding of how to write effective and efficient programs for shared-memory multiprocessor systems essential.

The challenge here arises from the fact that the computations of the different cores can affect each other, despite them ensuring an execution order that is consistent with the order in which the instructions appeared in the source code. Consider the pseudo-code example in Listing 3.1. Say P0 is being executed by hardware thread T0 and P1 is executed by hardware thread T1. By setting flag, T0 can pass a message to T1 via buf. Giving this no additional thought, one might expect that when the flag is set, T1 will read T0's message. The problem is that depending on the CPU architecture, it may be possible that flag is set, but T1 does not read T0's message. Instead, it reads the previous value of buf, even though both threads appear to execute in program order when considered on their own.

The reason this is possible is that maintaining the illusion of program order per core does not require side effects, such as writes to memory, to become visible in the same order in which they were issued. Accessing memory is usually a bottleneck, and in order for writes to memory not to stall execution, they may be committed to a store buffer, which can commit them to memory independently of the thread's main execution stream [OSS09]. If the CPU is able to load values from the store buffer too, the order in which the store buffer commits writes to memory is irrelevant for the illusion of program order, as it does not matter whether a value was obtained from the store buffer or main memory. On a uniprocessor system, the difference will never become apparent. But on a multiprocessor system, if writes are committed to main memory out of program order, another core's thread, which does not have access to any store buffer but its own, may see a different order of writes than the core that committed them - just like in the above example. Memory consistency models are the prime abstraction computer science uses for orchestrating and reasoning about such problems on shared-memory multiprocessor systems.

```
int buf = 0;
int flag = 0;

P0()
{
        WRITE(buf, 1);
        WRITE(flag, 1);
}

P1()
{
        int r1;
        int r2 = 0;

        r1 = READ(flag);
        if (r1)
                r2 = READ(buf);
}
```

Listing 3.1: The message passing pattern.

3.1. Defining Memory Consistency Models

Sorin et al. [SHW11] define memory consistency models as follows: "A memory consistency model, or, more simply, a memory model, is a specification of the allowed behavior of multithreaded programs executing with shared memory". This is a very general definition and applies to several layers in the computing stack, e.g. programming languages and instruction set architectures [AG96]. The "allowed behavior" enables us to put memory models on a spectrum from weak to strong according to how many states in a program's state space they permit. As memory models get stronger so do the constraints on the state space. Memory models can take shape in prose (informal) and mathematics (formal). Depending on context, the term "memory model" can refer to either, as both (attempt to) capture the same underlying behavior.

Shared-memory multiprocessor architectures came to be before programming languages supported them. Given the (sometimes) closed-source nature of CPU development, it was now up to academia to precisely capture the behavior of these new kinds of CPUs in formal models, as any prose CPU documentation falls victim to ambiguities and interpretation. Not only do such models help reason about the architecture, but they might also reveal bugs, as Alglave et al. show [AMT14].

Formal memory modeling has seen several approaches over the years. All of these approaches (or frameworks) aim to capture the behavior of a multiprocessor system, using a mathematical model. Given a piece of code, usually referred to as a litmus test, the model can be used to determine what executions (or outcomes) for that litmus test are permitted or forbidden.

Two schools of thought have established themselves over the last few decades. There are **axiomatic memory models** which are an umbrella term for execution-based frameworks for determining allowed and forbidden behavior by applying its axioms given interleaving of memory accesses [SFC92] [Sar+11]. And there are **operational memory models** which are an umbrella term for event-based memory models defined with an abstract machine, thereby trying to model the state of the shared-memory system [BP09] [AMT14].

3.1.1. Formal Memory Models

In the following, we give an overview of formal memory models.

SPARC One of the first published formal memory models of a computer architecture was the SPARC memory model, which happens to be a vendor-created one [SPA92]. It later served as an example for the first axiomatic formalization of a real shared-memory architecture [SFC92].

DEC Alpha A formalization of early DEC Alpha processors' behavior soon followed in the form of alpha consistency [AF94], using the framework proposed by Attiya et al. [Att+93]. Alpha's memory model is famously weak and in some cases does not even order dependent instructions as pointed out by the Linux kernel documentation in [Ling].

Intel X86 The formal memory model for Intel's X86 architecture is called total store order (TSO) and is based on the SPARC memory model. It is a prime example of a strong memory model as only allows reads to be ordered ahead of writes. All other read-and-write combinations are preserved, regardless of any dependency being involved. It was formalized in both an axiomatic and an operational memory model in [OSS09].

IBM Power IBM Power received an axiomatic memory model in [Alg+12] and an operational memory model [Sar+11].

ARM As of today, ARM maintains its own axiomatic memory model [Alg+21].

Formal programming language-level memory models exist too, with the Java memory model [MPA05] and the C11 and C++11 memory model [Bat14] being the prime examples. Others include the OCaml memory model [DSM18] and of course the Linux-kernel memory model [Alg+18].

Many of the aforementioned memory models can be executed in a memory model simulator. For instance, the herdtools7 suite allows users to generate litmus tests and execute them with a selected memory model [Alg10] [AMT14].

3.1.2. Informal Memory Models

Now, memory model simulators work well for litmus tests but are not able to handle large codebases such as the Linux kernel. This is where informal memory models come into play. An informal memory model is a piece of prose aimed at the general programmer.

Formal memory models aim to capture the dependency orderings CPU architectures provide at a mathematical level. Informal memory models give prose descriptions of these dependency orderings which are reminiscent of the pipeline hazards we described in §2.1. For example, before ARM maintained its own formal memory model, it informally defined an address dependency as follows [Arm09]: "Where the value returned by a read is used to compute the virtual address of a subsequent read or write (this is known as an address dependency), then these two memory accesses will be observed in program order. An address dependency exists even if the value read by the first read has no effect in changing the virtual address (as might be the case if the value returned is masked off before it is used, or if it had no effect on changing a predicted address value)." Intuitively, this definition makes sense, but it hardly suffices for use in a mathematical model. That would require a more precise definition of what it means for a value to be "used to compute [a] virtual address".

3.2. The C11 Memory Model

The C11 language standard introduced concurrency to the C programming language. The informal memory model is documented as part of the C standard section on `<stdatomic.h>` [WG118] [cpp] and has been formalized by Batty et al. [Bat14]. In C11, memory locations that are shared among threads are denoted with atomic variables. Accesses to atomic variables are synchronized based on the memory order that is attached to the access. C11 offers five memory orders, enabling different levels of consistency, which we will discuss now.

3.2.1. Sequential Consistency and Relaxed Consistency

By default, atomic accesses will be sequentially consistent and will implicitly have memory_order_seq_cst attached to them. Sequential consistency is the strongest memory order C11 has to offer, and is by no means a novel concept. Leslie Lamport defines a sequences of accesses to shared memory to be sequential consistency if "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" [Lam79].

This means that the order in which each individual threads executed its stream of instruction is specified—it is the order in which they appear in the programme—but the order in which the threads run is not. Threads could take turns after each running an instruction or only switch upon completion of their instruction stream.

Sequential consistency, while easy to grasp, does so at the expense of unrealized performance optimizations, as it requires instructions to execute in program order. That is why C11 allows performance-conscious programmers to encode their minimal ordering requirements into their code by specifying the desired memory order, giving compilers and CPUs more freedom for optimizations.

memory_order_relaxed takes this to an extreme and does not impose any ordering requirements whatsoever. Since it is attached to an atomic variable, the only guarantee it provides is atomicity. Given that the C language supports several CPU architectures with different guarantees for relaxed ordering, the effects of using relaxed semantics vary, depending on the CPU architecture being used.

3.2.2. Acquire and Release Semantics

memory_order_acquire can only be attached to atomic read operations. It stipulates that all reads and writes which are program-ordered after the acquire operation in the current thread must run after the acquire operation executes. It does not specify the order of any operations that come before the acquire operation in program order.

memory_order_release can be thought of as an inverse memory_order_acquire. It can only be attached to atomic write operations and specifies that all reads and writes that are program-ordered before the release operation in the current thread must be completed before the release operation executes. Again, it does not impose any constraints on the order of operations that come after the release operation in program order.

A memory_order_release in a given thread A and a memory_order_acquire operation in a different thread B, which reads the value released by thread A, can be paired together for release-acquire ordering. In that case, B is guaranteed to see all load and store operations that are program-ordered before the release operation in thread A. This guarantee does not hold for threads other than A and B, and it only takes effect when A and B access the same atomic variable.

memory_order_acq_rel provides these guarantees for read-modify-write operations and will not be discussed further. memory_order_acq and memory_order_release provide an improvement over memory_order_seq_cst in that they only impose ordering requirements on certain regions of code. However, within these regions, all instructions are ordered, no matter if they affect the acquire or release operation.

3.2.3. Consume Semantics

memory_order_consume, in theory, appears to be the best of all memory orders. When attached to an atomic read operation, it only requires that all dependent operations in the consuming thread happen after the consume operation. It behaves like memory_order_acquire except that it only orders dependent instructions. So, just like memory_order_acquire, memory_order_consume can be paired with a memory_order_release operation. The guarantees for this release-consume ordering are identical to those of release-acquire operations except that now only dependent instructions are ordered.

Now, memory_order_consume is special in the sense that it does not exist. Modern compilers do not implement memory_order_consume, but simply promote it to memory_order_acquire [cpp]. Since the memory_order_acquire guarantees subsume those of memory_order_consume, this will not lead to undesired behavior, but it will restrict compilers as well as CPUs in the optimizations they can make, which is exactly what the programmer wanted to avoid, as they otherwise would not have used memory_order_consume.

3.3. The Linux-Kernel Memory Consistency Model

The Linux kernel, although technically written in C, defines its own memory model, the Linux-Kernel Memory Consistency Model (LKMM). Having its origins in 2006, the Linux kernel introduced concurrency long before it made it into the C standard [How06]. In fact, the Linux kernel deviates so far from the C standard, that one could consider it a dialect of C. In the following, we will refer to this as Linux C. Linux C is a careful balancing act, as it provides its own implementations for features in and outside of the standard, but is still being compiled by compilers which strictly think in terms of the C standard and usually rely on their own implementation of standard features.

3.3.1. The Informal LKMM

The LKMM started as an informal memory model in a document called memory-barriers.txt [Ling], which, despite being nearly two decades old [How06], is still being used today.

In its most current form, the LKMM defines two macros for accessing shared memory: READ_ONCE() and WRITE_ONCE(). Both are akin to a C11's atomic memory_order_relaxed loads and stores, respectively, with the major difference that a READ_ONCE() can head dependency chains, imposing ordering constraints on subsequent dependent loads and store like memory_order_consume would do. Both are designed to be immune against compiler optimizations that would undermine their guarantees of ordering or atomicity. Any memory access using either the READ_ONCE() or WRITE_ONCE() macro is called a marked access.

Data Dependencies and Address Dependencies

The informal LKMM stipulates that all dependent memory accesses will be issued in order. This matches what we call the "most basic kinds of dependency orderings" in §2.1. Since most CPU architectures supported by the Linux kernel respect these dependency orderings, what the informal LKMM and Linux C, therefore, try to do is to ensure that the ordering of accesses is "outsourced" to the architecture. If the informal LKMM were to promise more ordering than architectures provide out of the box, it would have to insert costly barrier instructions, resulting in performance overheads at runtime, which is something the Linux kernel wants to avoid. In fact, performance is the main reason why the Linux kernel did not switch to C11 atomics and their memory model once they became available [Tor12].

Based on the guarantees the informal LKMM makes right now, the only thing left for the Linux kernel to do is insert barriers for architectures that potentially will not order dependent instructions, i.e., DEC Alpha [Ling] and to ensure that the code that is supposed to be ordered arrives at CPU s.t. that they can still infer the dependency. As it turns out, that is not a simple task in the presence of "big bad optimizing compiler[s]" [Alg+19].

Control Dependencies

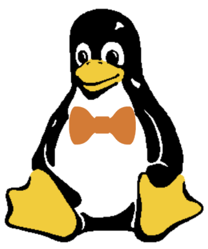
Given the Linux kernel's intention to match the ordering CPU architectures provide out of the box, the presence of control dependencies in the informal LKMM does not surprise. This once again marks a reason why the Linux kernel did not go with C11 atomics as they became available. They have no notion of control dependencies.

The informal LKMM only claims that read-to-write control dependencies are ordered. These look like Listing 3.2, where a branch condition depends on a marked read access and one of the branches contains a marked write access. Again, to avoid compiler optimizations undermining the guaranteed ordering, it is imperative to use marked accesses.

```c
r1 = READ_ONCE(a);
if (r1)
        WRITE_ONCE(x, 42);
```

Listing 3.2: A Control Dependency.

3.3.2. The Formal LKMM



In 2018, the informal LKMM received a formal counterpart in [Alg+18]. The formal LKMM is an axiomatic memory model that can be executed by the herd7 memory model simulator. In the Linux kernel source tree, the two informal LKMM and the formal LKMM have been separated. The informal LKMM lives in Documentation/memory-barriers.txt and is merely comprised of a text document, whereas the formal LKMM lives, together with all its litmus tests and separate documentation, in tools/memory-model/.

Above, you can see "formal Tux," the Linux kernel mascot in formal attire. Formal Tux originally helped us represent the formal LKMM in our talk at the Linux Plumbers conference [EH22] but has since grown to be somewhat of a mascot for our work. Formal Tux is based on the original GIMP drawings of Tux done by Larry Ewing [Ewi].

3.3.3. Contrasting the Informal and the Formal LKMM

The reason we chose to strictly differentiate the informal and the formal LKMM is that the informal LKMM strictly subsumes the formal LKMM in terms of ordered executions. In other words, whenever the formal LKMM predicts ordering, it matches the informal LKMM. However, this does not hold the other way around. This is a result of the formal LKMM trying to match what a CPU would see at runtime and being designed with the intention to be executable in the herd7 tool, thereby also inheriting its limitations [Ste22].

The informal LKMM and the formal LKMM significantly differ in their definition of control dependencies. For a control dependency to exist, the formal LKMM requires that a memory write "syntactically lies within an arm of an if statement [...] (or similarly for a switch statement)" [Linb] whereas the informal LKMM is less constraining and only requires a write to be within the scope of a branch that depends on a READ_ONCE() [Ling]. In the latter case, a write may be located in a function several function calls deep, i.e., not "syntactically [...] within the arm of an if statement" [Linb]. The formal LKMM's requirement that a write must lie within the branch also causes it to miss control dependencies such as the one in Listing 3.3.

```c
r1 = READ_ONCE(x);
if (r1 == 0)
        smp_mb();
WRITE_ONCE(y, 1);
```

Listing 3.3: A Control Dependency which the formal LKMM does not recognize [Linf].

4. Program Analysis

Program analysis is a means to ensure dependable systems. According to Brian Randell [Ran00], the term "dependability" is composed of several attributes, threats, and means. In the following, we concentrate on the latter two.

4.1. The Faul-Error-Failure Model

There are three threats to the dependability of a system: faults, errors, and failures.

Fault A fault is a compile-time concept. It is an error in the source code that is a result of a human mistake. A fault may lead to an error. We will use the term "bug" as a synonym.

Error An error is usually the result of a fault. It is a runtime concept and means that the system entered a state which may lead to a failure.

Failure A failure occurs once an error becomes observable. The system deviates from its specification.

Usually, a failure is the result of an error, which is the result of a fault, which is the result of a human mistake. We will abstain from extending the definition of a fault further, e.g., to include hardware malfunctions, as done in [Ran00] since they are not relevant to this work.

4.2. Means for Countering the Threats to Dependability

To identify and counter these threats, we can employ program analysis of which there are two kinds: static analysis and dynamic analysis. When employing either program analysis technique, one will face the metrics of soundness and completeness as well as the ensuing problems of false positives and false negatives.

4.2.1. Viewpoints on Measuring the Capabilities of Program Analyses

The definitions of soundness and completeness, as well as the ensuing definitions of false positives and false negatives, are a result of two opposing viewpoints. Either the analysis tries to prove that a program has a certain property, in that case, a bug-free program constitutes a positive, or the analysis tries to detect a certain kind of bug, then, a faulty program constitutes a positive [Mey19]. For our work, we take the latter viewpoint, bringing us to the following definitions of soundness and completeness.

4.2.2. Soundness and Completeness

A bug checker is sound if it reports every bug in the program. A bug checker is complete if every report is a bug. The perfect bug checker, therefore, would be sound and complete. Neither property implies the other. For instance, a bug checker that simply marks every program as bug-free would be complete as it never reports a bug when it should not—it reports no bugs at all—but it would not be sound (and would make its value questionable).

4.2.3. False Positives and False Negatives

The terminology of true/false positives/negatives follows from our viewpoint on (un)soundness and (in)completeness.

- A true positive is the bug checker reporting a real bug.
- A true negative is the bug checker not reporting a bug when there is none.
- A false positive is the bug checker reporting a bug when there is none.
- A false negative is the bug checker not reporting a bug when there is a bug.

4.3. Static Analysis

The term static analysis describes program analyses that are performed without executing the program. Static analysis may be run separately from or as part of program compilation.

4.3.1. Using LLVM for Static Analaysis

LLVM's compiler pass infrastructure provides a means for performing static analysis as a program is being compiled. Any LLVM compiler pass which does not modify the program is considered an analysis pass. It may report its results to the user or aid other transformation/analysis passes in their work.

Static Dependency Analysis

Static analysis can be used for identifying dependencies in a program. However, certain information such as pointers only becomes available at runtime, constraining any static analysis in its capabilities. It is not possible for static analysis to fully capture something that is dynamic, like dependency orderings for instance. Traditional static analysis counters this by redefining the notions of dependencies such that they match the capabilities of static analysis. Control dependencies are therefore reduced to reachability in a control flow graph, whereas at runtime, they are a data-flow problem.

Kennedy and Allen define control dependencies via a notion of post-dominance. Post-dominance is defined as: "A node V is post-dominated by a node W in G if every directed path from V to STOP (not including V) contains W" [Fer87]. In other words, post-dominance means that a node must be traversed on the way to the (function) exit. It is, therefore, not conditional. From that, a definition of control dependencies follows: "A statement y is said to be control-dependent on another statement x if (1) there exists a nontrivial path from x to y such that every statement z != x in the path is post-dominated by y, and (2) x is not post-dominated by y" [KA01]. This is a broader definition than that of the Linux kernel as it does not require a READ_ONCE(), a dependency chain into a branch instruction, and a WRITE_ONCE(). This definition does not match either of the LKMMs.

Consider Listing 4.1. LLVM arranges the IR such that the WRITE_ONCE() and the function return end up in the same basic block. As a result, the basic block containing the WRITE_ONCE() trivially post-dominates every basic block in the control-flow graph, including the one that contains the condition that depends on the READ_ONCE(). However, both LKMMs disagree. The formal LKMM sees that there is a WRITE_ONCE() which "syntactically lies within an arm of an if statement" [Linb] and the informal LKMM sees that a READ_ONCE()-dependent branch instruction will be emitted at runtime, ordering the WRITE_ONCE() against the READ_ONCE(). Both LKMMs agree that this control dependency would be ordered, contradicting the post-dominance definition of control dependencies.

4.4. Dynamic Analysis

The term dynamic analysis describes program analyses that are performed as the program is being executed. Dynamic analysis may rely on static analysis for its instrumentation, i.e. the insertion of additional code which triggers or helps the dynamic analysis at runtime.

```c
int *x, *y;

int foo()
{
        /* More code */

        loop:
                /* More code */

                if(READ_ONCE(x)) {
                        WRITE_ONCE(y, 42);
                        return 0;
                }

                /* More code */

                goto loop;

      /* More code */
}
```

Listing 4.1: An ambiguous control dependency.

4.4.1. The Kernel Memory Sanitizer KMSan

One example of a dynamic analysis tool is the Kernel Memory Sanitizer (KMSan), which describes itself as a "dynamic error detector aimed at finding uses of uninitialized values [in the Linux kernel]" [KMS]. KMSan is a Linux kernel-specific variant of the more general LLVM MemorySanitizer. What it considers a bug is based on the C standard's definition of undefined behavior.

For instance, the function call to foo in Listing 4.2 receives a function argument that has not been initialized. Such bugs are considered undefined behavior by the C standard and should be reported by KMsan.

For identifying what memory has been initialized, KMSan maintains a large memory mapping [KMS]. This memory mapping relies on the concept of shadow memory, where each memory byte of kernel memory is allocated a metadata byte of shadow memory for maintaining its initialization status. Shadow memory cannot be accessed by users. KMSan through shadow memory, KMSan can propagate uninitialized memory.

```c
int a;
int b;
b = foo(a); // Undefined behavior
```

Listing 4.2: Passing uninitialized function arguments into a function call is considered undefined behavior by the C standard. KMSan tries to detect such errors [KMS].

This is referred to as poisoning. For instance, the assignment of 0xff in Listing 4.3 initializes a and its shadow memory. b is uninitialized and KMSan is again able to track the uninitialized state in b's shadow memory. When a and b are combined with a logical OR, KMSan is able to perform an identical operation on the shadow memory. c's shadow memory therefore reflects that it is only partially initialized. The logical OR was only able to unpoison some of the shadow memory.

```c
int a = 0xff; // i.e. 0x000000ff
int b;
int c = a | b;
```

Listing 4.3: An example illustrating KMSan's shadow memory [KMS].

The value poisoning, or more generally maintaining the shadow memory, is implemented through compiler instrumentation, i.e. dedicated calls to KMSan for every relevant memory access, which are inserted during compilation. As a result, KMSan comes at the cost of a significant runtime overhead through its shadow memory as well as its instrumentation.

4.5. Symbolic Execution

Symbolic execution marks a particularly interesting program analysis technique, as it could be considered a hybrid between static and dynamic analysis. King et al. introduce symbolic execution as follows: "Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols" [Kin76].

Symbolic execution is static in the sense that it does not execute the program, yet it is dynamic in the sense that it is able to capture all possible executions the program might take at runtime. The latter marks symbolic execution's biggest advantage and drawback at the same time. Being able to capture every possible execution makes it incredibly powerful for program analysis, but makes it face an exponential state space at the same time. On top of that, any symbolic execution that does not start at the beginning of the program must guess values for the program context, e.g. variables. As Gritti et al. point out in [Gri+20], if the value that is guessed determines the termination of a loop, the symbolic execution might get stuck in an infinite loop. And if the value heads a data dependency, guessing the wrong value might break that dependency. Since guessing values can lead to invalid program states, symbolic execution is often infeasible for larger programs where their size would make guessing values necessary.

4.5.1. Concolic Testing

There exist hybrid approaches for program analysis involving symbolic execution. For instance, concolic testing [Sen07] sees a program being executed concretely with default inputs. By tracking the branch conditions as symbolic constraints, after the execution has been completed, new inputs can be generated for new program coverage. Such an approach may be used for test case generation [SMA05].

4.6. angr

angr (stylized in lower-case) is a binary analysis framework [Sho+16]. Not unlike what LLVM does for toolchain developers, angr aims to provide users with the tools to perform various program analysis techniques, including static and dynamic analysis as well as symbolic execution. angr works on the compiled binary representation of a program.

angr was designed with usability in mind and is still actively maintained. Its other design goals were cross-architecture support, cross-platform support as well as support for different analysis paradigms. It tries to reuse existing frameworks and libraries when possible. Users can interact with angr through an interactive Python shell (IPython).

Binaries are loaded into angr with CLE, a recursive acronym for CLE Loads Everything, enabling its cross-platform support. For a given binary, angr will maintain several different representations of that binary, depending on the analyses users want to perform. This includes a disassembled representation, generated with the capstone disassembly framework, and an intermediate representation in Valgrind's VexIR, generated with the libVEX IR lifter.

The SimuVEX module allows angr to maintain program states, called SimStates. SimStates are an abstraction that includes everything from current register values, symbolic memory, and concrete memory to open files and logs. All of this information is made accessible to users through SimState plugins which they can interact with.

Data, e.g. register values, are represented by expressions provided by the Claripy module. Expressions can be resolved to real values in the respective data domain at any time by using one of Claripy's backends. For instance, for symbolic data, Claripy will use its Z3 backend.

The entry point for program analyses is the Project class, representing a binary that was loaded into anger. Project objects allow the user to call existing angr analyses, e.g. control-flow graph recovery, or run their own.

4.6.1. Symbion

As we pointed out in §4.5, symbolic execution faces the issue of state explosion and is limited whenever it begins at a point in the program that is not the beginning. To alleviate these problems, angr provides an interleaved symbolic execution mechanism called Symbion. Interleaved symbolic execution is a term coined by Gritti et al. [Gri+20] and describes a mechanism that is able to switch between symbolic and concrete executions on the fly, like Symbion.

Symbion's interleaved symbolic execution has three phases. The first phase sees the program being executed concretely until some point of interest, i.e. program counter, is reached. The second phase marks the switch to symbolic execution, starting at the point of interest, where the concrete environment is used as the base for the symbolic execution. The symbolic execution proceeds until a given target is reached. The third and final phase marks the switch back to concrete execution. Symbolic constraints on the variables are evaluated and re-synced with the existing concrete environment. Re-syncing the environment progresses the concrete execution from the point of interest to the target point.

This approach decouples the symbolic execution from the concrete execution and allows the symbolic execution to progress from any point with real inputs. The state explosion problem can be contained (or at least postponed) by not making the whole program state symbolic in the second phase but on the variables the programmer is interested in for their analysis.

4.7. Connecting Static and Dynamic Analysis with LLVM PC Sections

To let a static analysis inform a dynamic analysis of points of interest in the program, LLVM provides an annotation mechanism that allows users to save points of interest in the compiled binary. This mechanism is called LLVM PC Section metadata [LLVd]. PC Section metadata can be generated in the LLVM middle-end, i.e., at an LLVM IR level. Users may annotate individual instructions or functions with such metadata. Each annotation consists of a section name and an arbitrary number of constant data elements, e.g., integers. LLVM will ensure that all annotations with the same section name will end up in an equally-named section in the compiled binary. The section header in the compiled binary will inform users at which address a given section begins and how large it is. Users will need to parse the LLVM-generated sections, considering how much constant data they attached to the PC section. Each entry contains a program counter in the form of a relative program counter, hence the name PC, which identifies the instruction/function annotated in IR and the attached constant data. The relative program counters are defined as a signed integer that must be added to the address of the entry itself.

4.8. Fuzzing

Reaching any part of a program is non-trivial. If the question is whether the end of the given program can be reached, it boils down to the halting problem, which is undecidable [Tur37]. Yet, for software testing, achieving a high amount of coverage is still desirable. Symbolic execution faces the state explosion problem, but heuristics still exist to maximize coverage without solving the reachability problem. One such mechanism is coverage-guided fuzzing. A coverage-guided fuzzer is a testing mechanism that can generate inputs for the program under test. It can modify its inputs based on the achieved coverage to reach new program regions, i.e., achieving more coverage.

4.8.1. Fuzzing the Linux Kernel with syzkaller

syzkaller (stylized in lower-case) is a coverage-guided fuzzer for the Linux kernel [syzc]. syzkaller started as a Linux kernel-only fuzzer but has since been extended to support other operating systems such as Darwin/XNU or Windows. Syzkaller relies on program instrumentation to gain information on achieved coverage. Instrumentation is enabled through a configuration option in the Linux kernel's KConfig system and added as the Linux kernel is compiled.

The user-facing interface of the Linux kernel is its system calls. Syzkaller comes with its own language, syzlang, for describing the program inputs, in this case, a set of systems calls to the Linux kernel. Syzkaller maintains a so-called corpus of these inputs when testing a program. After the corpus is populated with some initial value(s), an input is picked from the corpus and modified by syzkaller, e.g., a bit is flipped. The program is executed with that modified input, and if it generates new coverage, the modified input is added back to the corpus. If it does not generate more coverage, the input is thrown away. syzkaller is continuously being run on the Linux kernel by syzbot and the developers maintain a publicly-accessible list of reported bugs [syza].

Part III.

Automatic Detection of Broken Dependency Orderings in the Linux Kernel

5. Broken Dependency Orderings in the Linux Kernel

As discussed in §1, we investigate if and how compiler optimizations break the dependency orderings defined by the Linux kernel memory model. Given the informal definitions of dependency orderings outlined in §3.3, we require more robust definitions of the dependency orderings and when a dependency counts as broken. These definitions should be formulated such that a static analysis can implement them.

5.1. Decomposing Dependency Orderings in the Linux kernel

To better understand the notion of dependency orderings, we introduce the following abstractions.

5.1.1. Dependency Chains

The notion of dependency chains is central to the Linux kernel's dependency orderings. Without them, we would not be able to infer a dependency in the first place. All three LKMM dependency orderings rely on dependency chains. Once again, there is some ambiguity to be resolved with the term dependency chain.

Singly linked dependency chain

One viewpoint is that there only exists one dependency chain per dependency. From that viewpoint, dependencies would be uniquely identified by their dependency chain. If two dependencies of the same kind have the same beginning instruction and the same ending, but one dependency takes a different path to that ending, the dependencies are not considered equal.

Multiply linked dependency chain

The second viewpoint extends §5.1.1 s.t. it can account for different paths between the same beginning and the same ending. In this case, dependencies are uniquely identified by their beginning and ending.

This gives way to the definition of partial dependencies. Partial dependencies are those where the dependency chain does not run on every control flow path from the beginning to the ending instruction. For instance, the dependency chain may depend on a conditional. If the conditional is true, the dependency holds; if the conditional is false, the dependency chain is missing links that would otherwise be introduced in the true branch of the conditional, and no dependency exists.

Partial dependencies imply the existence of complete dependencies, where the dependency chain runs on every possible control-flow path from the beginning to the ending, and null dependencies, where there is no dependency chain connecting the beginning and ending instructions on any control-flow path even though there should be. We do not choose to call null dependencies broken dependencies since dependencies count as broken depending on the conversion performed by the compiler, not the dependency type. We discuss this further in §5.2.

5.1.2. Defining Dependency Chains

If we consider instructions as values, dependency chains consist of all values that recursively depend on their beginning. Dependency chains are not constrained by any kind of programming language construct perse and may run in and out of function calls, across translation units, and into inline assembly.

5.1.3. Data Dependencies and Address Dependencies

Data dependencies consist of a dependency chain. The dependency must run from a READ_ONCE() into the data operand of a WRITE_ONCE().

Address dependencies consist of a dependency chain. The dependency chain begins with a READ_ONCE() and may end at the return value of a READ_ONCE() if its source operand is part of the dependency chain or a WRITE_ONCE() if its destination operand is part of the dependency chain.

Given the above definitions, one could argue that address dependencies are just a specialization of data dependencies. An address is after all data. Any address dependency implies a data dependency, yet not every data dependency implies an address dependency. However, this is not the viewpoint the Linux kernel takes, where data dependencies and address dependencies are considered disjoint. Data dependencies end in the data operand in a WRITE_ONCE(), and address dependencies end in the destination operand of a WRITE_ONCE() or the source operand of a READ_ONCE().

5.1.4. Control Dependencies

Control dependencies consist of two components. The first is a dependency chain, running from a READ_ONCE() into a branch instruction, e.g., if condition, switch condition, or loop condition. There are no requirements for the branch condition itself, apart form it having to depend on the READ_ONCE(), heading the dependency chain. Even if the branch depends solely on an address, to evaluate that condition, the address would have to be converted to a boolean. Loosely speaking, that would make it a data dependency, but to avoid any confusion with the LKMM's definition of data dependencies, which are required to end with a WRITE_ONCE(), we will avoid using that term in this context. The second component is a WRITE_ONCE(), which lies in the scope of the data-dependent branch instruction. The scope of a branch condition ends at the point where all paths in the control flow graph that start at the branch condition meet again. This point does not necessarily have to exist. For instance, consider the control dependency depicted in Listing 5.1. The if branch and the implicit else branch never meet, therefore, the dependency does not get resolved.

```
restart:
r1 = READ_ONCE(foo);
if(!f1)
        goto restart;
// More code
```

Listing 5.1: A simple control dependency that does not get resolved.

5.1.5. Syntactic and Semantic Dependencies

Now, it is important to point out that the above definitions are static definitions of LKMM dependency orderings. They describe syntactic dependencies [Tor14] [Ste21]. A syntactic dependency might also be a semantic dependency but not vice versa. Semantic dependencies are those that were intended by the programmer and exist at runtime. Syntactic dependencies are a strict superset of semantic dependencies. Semantic dependencies reflect the programmer's intent and cannot be determined statically.

5.2. How Dependency Orderings in the Linux Kernel Could Be Broken

Given the definition above, we must know how the dependencies they define can be broken. This happens by transforming the dependency from one type, i.e., full, partial, or null dependency, to another. Below, we will highlight the transformations that we consider dependency-breaking. This is done to build an intuition for how dependencies may be broken. Since the aim of this work is to investigate just that, this description is by no means exhaustive.

5.2.1. Breaking Address and Data Dependencies

Given the similar structure of data and address dependencies, in our view, they share the same kinds of potential breakages.

Full to Partial Conversion

Consider the address dependency in Listing 5.2. If a conditional were to be introduced on the dependency chain, as seen in Listing 5.3, that would break the dependency. It would allow weakly-ordered architectures to speculate the values within the conditional, making it possible for the second READ_ONCE() to be executed before the first READ_ONCE() even though the programmer and the LKMM did not intend this.

```
r1 = READ_ONCE(*x);
y = &r1[42];
r2 = WRITE_ONCE(*y, 10);
```

Listing 5.2: An address dependency at risk of being transformed into a control dependency [Dea20].

(Full or Partial) to Null Conversion

The most trivial way any full or partial data dependency can be broken is by transforming it into a null dependency by breaking the dependency chain on all control-flow paths connecting the data dependency beginning and ending. Consider the dependency in Listing 5.4 where breaking the dependency chain might result in something as depicted in Listing 5.5 if the compiler can deduce the value of the address being written to. Of course, we cannot expect broken dependency orderings to present themselves in such a simple fashion, and we discuss some more intricate cases in §6.3.

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

Listing 5.3: A control dependency resulting from an address dependency transformation [Dea20].

```
r1 = READ_ONCE(*x);
r2 = &r1[42];
WRITE_ONCE(z, r2);
```

Listing 5.4: A data dependency at risk of being transformed into a null dependency.

5.2.2. Breaking Control Dependencies

Control dependencies imply a dependency chain from a READ_ONCE() into a branch condition. That makes them susceptible to the same breakages outlined for data/address dependencies outlined in §5.2.1 and §5.2.1. All those transformations may also be applied to the dependency chain into the conditional. Changing the scope of a control dependency or removing it altogether is the second way in which control dependencies can be broken. We highlight a few known cases of how such transformations may materialize, with more examples yet to be explored.

Identical Writes in Both Control Dependency "Legs"

As has been pointed out several times in the Linux kernel community [Dea21] [Lina], if a control dependency contains two identical WRITE_ONCE() instructions in both "legs," as seen in Listing 5.6, the compiler may remove one of them and hoist the other one out of the conditional as depicted in Listing 5.7.

Syntactical Branch Condition

If the branch condition is merely syntactic, the compiler may remove the conditional branching altogether as it will be able to determine statically whether the condition will either be always true or always false. Listing 5.8 gives an example of a trivial syntactical branch condition. A compiler would then be within its rights to remove the

```c
r1 = READ_ONCE(*x);
r2 = r1[42];
WRITE_ONCE(z, 21);
```

Listing 5.5: A null dependency resulting from a full data dependency.

```c
x = READ_ONCE(*foo);
if (x > 42) {
       WRITE_ONCE(*bar, 1);
       frob();
} else {
       WRITE_ONCE(*bar, 1);
       twiddle();
}
```

Listing 5.6: A control dependency with two identical writes in its branches [Dea21].

branch as shown in Listing 5.9. Such branch conditions may not always be obvious to programmers as they can hide behind arithmetic and constants. In Listing 5.8, the constant MAX may also be defined differently based on the target architecture, making the existence of a semantic dependency dependent on the Linux-kernel configuration.

Transformation of a Control Dependency to an Address Dependency

Will Deacon pointed out to us that a control dependency such as the one shown in Listing 5.10 may also be transformed into an address dependency like the one shown in Listing 5.11. While this would not break ordering, it would still prove to be interesting information for CPU architects he said.

5.3. Requirements for a Dependency Checking Mechanism

In the following, we will outline the perfect dependency-checking mechanism for catching dependency breakages such as (but not limited to) those outlined in §5.2. We envision that such a dependency-checking mechanism would appeal to at least five groups of users:

1. **Linux-Kernel Memory Model Maintainers** could use such a dependency-checking mechanism to get an estimate of the extent to which the LKMM is being un-

```c
x = READ_ONCE(*foo);
WRITE_ONCE(*bar, 1);
if (x > 42) {
        frob();
} else {
        twiddle();
}
```

Listing 5.7: A control dependency where two writes were merged and hoisted out of the branches [Dea21].

```c
#define MAX 1

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

Listing 5.8: A syntactical control dependency [Dea21].

dermined by modern optimizing compilers. Based on that, they could make informed decisions about changing the LKMM or introducing additional protection mechanisms for LKMM primitives.

2. **Linux Kernel Hackers** could use such a dependency-checking mechanism to ensure that the dependency orderings in their patches are being respected by modern optimizing compilers.

3. **Compiler Engineers** could use such a dependency-checking mechanism to ensure that the optimizations they are working on respect the LKMM.

4. **Rust for Linux Maintainers** could use such a dependency-checking mechanism to make informed decisions about the Rust memory model in the Linux kernel [Fen23]. Apart from Linux C, the dependency-checking mechanism would have to support Rust as well in this case.

5. **Security Researchers** could use such a dependency-checking mechanism as part of their continuous testing infrastructure for the Linux kernel for getting information about memory-model-related bugs in current Linux kernel builds.

The dependency-checking mechanism would ideally run as part of compiling the

```c
#define MAX 1

x = READ_ONCE(*foo);
WRITE_ONCE(*bar, 1);
```

Listing 5.9: A syntactical control dependency where the branch has been optimized away [Dea21].

```c
x = READ_ONCE(*ptr);
if (x & 1)
    WRITE_ONCE(z, 42);
else
    WRITE_ONCE(y, 21);
```

Listing 5.10: A control dependency which may be transformed into an address dependency.

Linux kernel. It could identify the dependency orderings on its own and determine with certainty whether they were broken or not. Given this potentially vast and diverse user base, such a dependency-checking mechanism must be developed with usability in mind. It should keep false positives to a minimum, provide several modes based on the interests of a given user group, and ideally provide suggestions out of the box.

5.4. Overview of Our Work

Given the requirements above, we propose the LKMM Dependency Checker (or DepChecker for short), an end-to-end dependency-checking mechanism, as part of the clang compiler. The DepChecker can be split into two components: a static component, the StatDepChecker, and a dynamic component, the DynDepChecker.

```
x = READ_ONCE(*ptr);
WRITE_ONCE(y[x & 1], 42);
```

Listing 5.11: An address dependency resulting from a control dependency transformation.

6. The Static Dependency Checker

The StatDepChecker, as the name suggests, only runs at compile time. As a result, it can only annotate syntactic dependencies. It cannot infer the programmer's intent by simply looking at the syntax. The StatDepChecker has three responsibilities:

1. Identify LKMM dependency orderings.
2. Check whether compiler optimizations broke the identified LKMM dependency orderings and breakages to the user.
3. If it cannot confidently determine whether a dependency got broken or not, it will delegate the dependency checking to the DynDepChecker.

6.1. Design

As we want to make the StatDepChecker a part of the Linux kernel compilation process, there are only two compilers to choose from. GCC, which is the Linux kernel's main compiler, and clang, for which Linux kernel support is being ensured by the ClangBuiltLinux Project [Cla]. We chose clang (LLVM) for its easy extensibility.

6.1.1. Tracking Dependency Chains

The DepChecker must keep track of the control-flow paths that may connect a dependency beginning and ending. To do so, we chose to have it traverse a module's control-flow graph in a breadth-first search, allowing it to track the different control-flow paths simultaneously, whereas a depth-first search would mean tracking them independently and keeping track of where they diverge. Tracking the dependency chain means that For every LLVM IR instruction type, the annotator must know how it may affect the dependency chain. Dependency chain values may be overwritten.

6.1.2. Tracking the Scope of Control Dependencies

Tracking control dependencies requires knowledge of how the control dependency gets resolved. Given our control-flow-based definition of control dependencies outlined in §3.3.1, identifying that point requires tracking when control-flow paths split and meet again. Every successor of a basic block in a control-flow graph marks a potential control-flow path. Control-flow paths stop when they run into a basic block that has been traversed by the BFS before and merge when they run into the same basic block.

6.1.3. The Three Stages of the StatDepChecker

The StatDepChecker works in three stages, which are pinned along the middle-end and backend of the clang compiler.

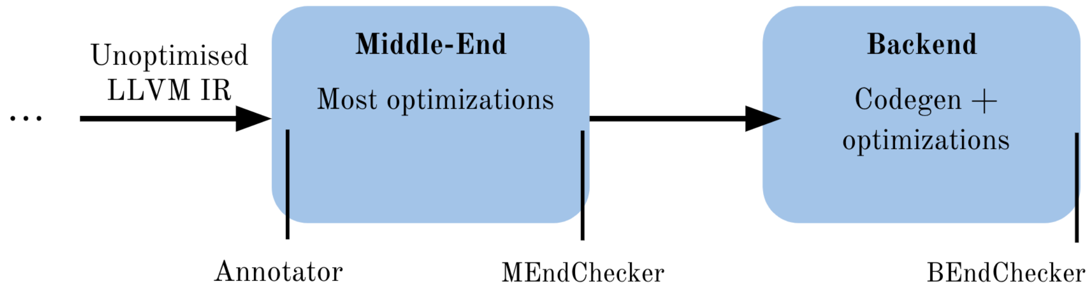


Figure 6.1.: The static dependency checker.

6.1.4. The Annotator

The annotator runs before most compiler optimizations in the LLVM middle-end. It gets called for every LLVM IR module and marks every dependency ordering it sees for the later stages to use. To do so, it must track the dependency chains for every potential dependency beginning. If it encounters a READ_ONCE() or a WRITE_ONCE(), it determines whether an address or data dependency exists by looking at the relevant operand. If the operand is part of a dependency chain, it annotates it as a full or partial dependency. If it encounters an instruction that splits control flow, it determines whether its condition value is part of a dependency chain. If it is, it starts tracking the scope of that conditional. Any WRITE_ONCE() encountered within the scope of that conditional is annotated and considered control-dependent on the branch and the READ_ONCE() it depends on.

6.1.5. Middle-End Checker

The middle-end checker runs after most optimizations in the LLVM middle-end. It functions similarly to the Annotator, except it knows which dependency chains to track since the beginnings were annotated. Every annotated beginning it encounters is considered broken until proven otherwise. The counterproof is made by tracking the dependency chain from the beginning in question until it encounters the annotated ending instruction and the respective operand being part of the dependency chain. It reuses the annotator's dependency chain tracking mechanisms for making the counterproofs. If a counterproof cannot be made for an annotated dependency, it is either printed to the user or delegated for further checking by the BEndChecker and/or the DynDepChecker if set up.

6.1.6. The Backend Checker

The BEndChecker is a replication of the MEndChecker in the LLVM backend. The MEndChecker is work done by Martin Fink. It does not work on the same IR the Annotator and the MEndChecker work on; it works the lower-level LLVM machine IR, which is a closer representation of the assembly and only contains a finite amount of physical registers, whereas LLVM middle-end IR has an infinite amount of virtual registers at its disposal [Fin23].

6.2. Implementation

The StatDepChecker is implemented in the form of three compiler passes, each of which is scheduled on a best-effort basis in the LLVM optimization pipeline through the callback functions provided by LLVM. Two of those compiler passes are implemented by us and can be viewed at: `https://github.com/PBHDK/mthesis-llvm-project/blob/main/llvm/lib/Transforms/Utils/LKMMDependenceAnalysis.cpp`. Our implementation is based on LLVM 17.

Dependency chains are implemented as an unordered set, as the insertion order does not matter for the DepChecker's purposes and fast membership checks are required. For a given dependency beginning, the StatDepChecker tracks all dependency chain values of all control-flow paths in a single set. This allows it to model full and partial dependencies via a multiply-linked dependency chain. Every dependency chain link has a level attached to it. The level keeps track of whether a given value is part of a dependency chain itself or instead points to a dependency chain value. This info is needed to determine when a dependency chain value is overwritten. If a store writes a value that is not part of the dependency chain and the destination points to a dependency chain value, the dependency chain value must be removed from the dependency chain. If the destination pointer is part of the dependency chain, but what it points to is not, the store instruction does not affect the dependency chain.

Every dependency is uniquely identified by its beginning and ending as well as the path in between. That allows the Annotator to construct a unique string ID for every dependency it identifies. For address and data dependencies, the IDs are defined as the location of the dependency beginning concatenated with the path taken from the beginning to the ending concatenated with the location of the dependency ending. The path taken is represented as all the locations of all the function calls and function returns where the dependency chain got passed in/out. In addition, the path taken is used to determine when a dependency ending was inlined. For control dependencies, the IDs look similar and are defined as the location of the dependency beginning concatenated with the path taken from the beginning to the branch instruction concatenated with the location of the branch instruction concatenated with the location of the location of the WRITE_ONCE(). For every dependency type, this representation is unique to a dependency. If two IDs were to match, it would imply the dependencies being equal. This can happen when a dependency beginning or ending gets duplicated, which the StatDepChecker accounts for.

The current implementation of the StatDepChecker tracks to-null conversions of address dependencies only. We had initially tried full-to-partial conversions of address dependencies by tracking a set of the union of all dependency chains as well as a set of the intersections of all dependency chains. That is a set that contains the values of all dependency chains, allowing us to overestimate partial dependencies by reasoning that there is some dependency chain connecting a beginning and an ending, as well as a set that only contains values that are present on all dependency chains (full). In the latter case, every value in the set is seen by every execution, as none of them are conditional. However, our implementation was incomplete, and in favor of prioritization, we decided to focus on to-null conversions and not differentiate between full and partial dependencies when annotating. The current implementation only tracks the union of all dependency chains for a given dependency.

We believe that supporting data dependencies would be as simple as adding a case to how volatile writes are handled in IR as well as introducing a data dependency ID type. This would require further testing, which we were not able to complete in time for submission of this thesis.

We have several work-in-progress control dependency implementations:

Attempt 1: tracking the scope w/o annotating branches Our first attempt for tracking control dependencies was to annotate them like address dependencies, i.e., only annotate the beginnings and endings, and require them to still be within the scope of any conditional that depends on the dependency beginning. This implementation led to false positives, as removing the branch in the case of a trivially true branch condition caused the dependency checker to report a broken dependency.

Attempt 2: only annotate the beginning and the dependent branch Our second attempt only saw us annotating the dependency beginning as well as the dependent branch. This approach only tracked dependency chains into conditionals and never identified any dependent WRITE_ONCE() access. It was, therefore, a deliberate overestimation. However, it fell victim to legal branch transformations that compilers might make in the absence of a WRITE_ONCE(). Further, kernel builds did not complete with this implementation, which we attribute to a bug that might be related to the amount of annotations being made.

Attempt 3: union of attempts 1 and 2 Now, our desired implementation is the union of attempts 1 and 2, meaning that the scope is tracked and the WRITE_ONCE() is identified but the branch instructions are also annotated. If the annotation of the branch instruction is not present anymore in optimized IR, we attribute this to a legal compiler transformation. Breakages will only be reported if the WRITE_ONCE() does not lie within the scope of the annotated conditional anymore or the dependency chain between the annotated beginning and the annotated conditional got broken. This would require us to merge our implementations of attempts 1 and 2 as well as further testing, which we were not able to complete in time for the submission of this thesis.

6.3. Evaluation

In this section, we will describe how we tested our StatDepChecker implementation as well as our results.

6.3.1. Test Cases

To ensure the core functionality of the StatDepChecker, we provide a set of dependency orderings written for testing purposes based on what McKenney et al. proposed for testing a memory_order_consume implementation [McK+17]. These test cases are written in Linux C and are part of the Linux kernel fork we use for this project. That means that they will be identified by the StatDepChecker as part of the Linux kernel compilation. If enabled through a command-line flag, after the Annotator runs its breadth-first search, it will artificially break the test cases to model transformations to null dependencies. Listing 6.1 shows an unbroken test case in Linux C and Listing 6.2 shows an approximation of how the broken test case would look in Linux C. The full list of test cases can be viewed at: `https://github.com/PBHDK/mthesis-linux/blob/main/proj_bdo/dep_chain_tests.c`

```c
static noinline int proj_bdo_rr_addr_dep_begin_simple(void)
{
    int *r1;
    int *r2;
    int r3;

    r1 = READ_ONCE(*x);

    r2 = &r1[8];

    r3 = READ_ONCE(*r2);

    return r3;
}
```

Listing 6.1: An unbroken artificial dependency ordering used for testing the StatDepChecker.

6.3.2. Random Testing

However, even with the test cases we provide, we came to the conclusion that the best test case we have is the Linux kernel. For robustness testing, we provide a Python script that is able to randomly generate Linux kernel configurations, build them, and gather the results along the way. It can track dependency breakages it has seen before and only collects new results. The random testing script can be viewed at: `https://github.com/PBHDK/mthesis-linux/blob/main/proj_bdo/random_search.py`.

6.3.3. Observed Dependency Breakages and Notable False Positives

In the following, we outline observed dependency breakages as well as notable false-positive reports that inspired additional work on the Dependency Checker. All of these reports have been observed during the development of the DepChecker and do not reflect the current output of the DepChecker for a Linux kernel default configuration. That output of course depends on the kernel version, the kernel configuration as well as the versions clang, and the StatDepChecker being used for compilation. To denote code locations, we use the following notation: source_file::line. All source files are relative to the Linux kernel source tree's root. Instead of inlining potentially complex chains of function calls, we will simply mark the topmost calls to the dependency beginning and ending with comments for easier reading, and we annotate the dependency beginning

```c
static noinline int proj_bdo_rr_addr_dep_begin_simple(void)
{
    int *r1;
    int *r2;
    int r3;
    volatile int bug_val1;
    volatile int* bug_val2;

    r1 = READ_ONCE(*x);
    bug_val1 = 42;
    bug_val2 = &bug_val1;

    r2 = &bug_val2[8];

    r3 = READ_ONCE(*r2);

    return r3;
}
```

Listing 6.2: An artificially broken dependency ordering used for testing the StatDepChecker. Since the test cases are broken on an IR-level, this is an approximation of what it would look like in C.

and ending with source code comments. All comments using the // syntax are made by us, comments using the /**/ syntax are part of the kernel code, and [...] denotes omitted code that is not relevant for inferring the dependency. Even if not every of the following dependency breakages represents a broken semantic dependency, they still count as trophies for the StatDepChecker, as it reported them to the best of its abilities. The following code reflects the current stable kernel version at the time of writing, which is version 6.5.7.

Broken Dependency in mm/ksm.c::2114

The dependency breakage in Listing 6.3 happens because of the stable_node->head = &migrate_nodes assignment. stable_node is part of the dependency chain and runs into a dependent function argument in the following line as part of the list_add(&stable_node->list, stable_node->head) call. It is the second function argument that we are concerned about. Since the compiler knows that stable_node->head must equal &migrate_nodes

replaces the value in the function call. As &migrate_nodes is not part of the dependency chain, there is now no dependency chain connecting the dependency beginning and the dependency ending, rendering the address dependency broken.

However, as it was pointed out on the Linux kernel mailing list [Hei22a], the Linux kernel got lucky in this case as the whole dependency is surrounded by a lock which gets acquired a few function calls earlier in the call stack. Due to mutual exclusion, no other thread will be able to observe the dependency breakage.

This case has been discussed and confirmed on the Linux kernel mailing list [Hei22a] and was part of our presentation at the Linux Plumbers Conference [EH22]. The current version of the StatDepChecker is not reporting this case. This may result from concessions in favor of avoiding false positives or the compiler simply not performing the dependency-breaking optimizations in the first place. We have observed a variance in dependency breakages as part of our testing.

Broken Dependency in fs/nfs/delegation.c::613

The dependency breakage in Listing 6.4 happens because of the use of a dependency chain value in an inequality comparison, namely delegation != place_holder_deleg where delegation is part of the dependency chain and place_holder_deleg is not. Whenever this inequality comparison evaluates to false, the compiler knows that the values must be equal, making it possible for it to replace one with the other. However, since only one of the values is part of the dependency chain, such a replacement breaks the dependency chain, as is the case here. In a sense, this may be considered an example of an address to control dependency transformation per [Dea20], as the breakage of the dependency chain is conditional on a place_holder_deleg and delegation being equal.

Again, this dependency breakage is protected by a lock, which we had not realized when discussing the case on the Linux kernel mailing list [Hei22b], and it does not strictly adhere to the documentation of rcu_dereference(), which points out that comparisons of values returned by rcu_dereference() against non-NULL values could lead to a compiler replacing one with the other [Link].

In this case, we were able to verify that with the current version of the StatDepChecker, the dependency is not being annotated because the dependency chain runs into a llvm.is.constant.i64 intrinsic call. To avoid false positives, the StatDepChecker does not annotate the dependency.

Broken Dependency in mm/memory.c::3878

This dependency breakage, shown in Listing 6.5, is reminiscent of the dependency breakage in Listing 6.4 except that the equality comparison enabling the compiler to break the dependency chain between folio and swapcache manifests itself more clearly in the source code this time. To the best of our knowledge, this dependency breakage does not represent a broken semantic dependency as it is yet again protected by a lock.

Broken Dependency in drivers/hwtracing/stm/core.c::1071

The dependency breakage shown in Listing 6.6 is an inverse of Listing 6.5, as it is an inequality comparison that enables the dependency-breaking optimization this time. Since there is a goto unlock at the end of the link != stm if-condition's only branch, reaching the code immediately after the if branch, implies that the condition must have evaluated to false. If the condition evaluates to false, link and stm must be equal, allowing the compiler to replace one with the other, again, thereby breaking the dependency chain. This case has been confirmed on the Linux kernel mailing list but is once again protected by a look as well as a control dependency [Hei23].

Broken Dependency in kernel/locking/lockdep.c::6457

Listing 6.7 shows another equality comparison of a dependency chain value with a non-dependency chain value that is causing the dependency breakage. In this case, it is the k == key comparison. This case has been confirmed on the Linux kernel mailing list and is yet again protected by a lock as well as a control dependency [Hei23].

A false positive in drivers/xen/pvcalls_front.c::796

Listing 6.8 marks a particularly interesting false positive, as it motivated our work on tracking dependency chains on several levels. The dependency chain starts with the req_id assignment. A get_request(call follows, which receives a pointer to req_id, i.e. a pointer to a dependency chain value. The DepChecker must now track the address req_id not as being part of the dependency but as pointing to the dependency chain. req_id gets overwritten in the function call, and as a result, the following READ_ONCE(bedata->rsp[req_id]. req_id) == req_id) does not depend on the READ_ONCE() that returned the first value of req_id.

False Positive in net/ipv6/ip6_fib.c::1563

Listing 6.9 is the only proof we have that avoiding backedges when traversing the control-flow graph is in rare cases not enough for the StatDepChecker's Annotator to avoid annotating dependencies through, i.e. those that require at least one loop iteration to exist. In this case, the compiler had not determined yet that at least one loop iteration was required to leave the first loop. As a result, the Annotator marked the two dependencies. In optimized IR, the code got transformed such that one loop iteration was required in IR to connect the dependency beginning and ending. Since the MEndChecker, like the Annotator, avoids backedges, it was not able to make the connection and falsely reported the dependency as broken.

6.3.4. How to Get Away With Breaking a Dependency

During testing, we discovered that there are ways where compilers can "get away" with breaking a dependency. This does not take anything away from the dependency breakage perse, but it does mean that if the kernel were run such that it would execute the dependency ordering in question, any reordering the CPU might decide to make would not matter.

Syntactic Dependencies

The most trivial breakage which is not considered dangerous is the breakage of a syntactic dependency. Syntactic dependencies might be as trivial as discussed in §5.2, but any dependency where the programmer does not expect ordering is a syntactic dependency. That means that the final say on dependency ordering is always up to the programmer. The programmer's intentions can never be inferred by merely looking at the syntax.

Dependencies protected by locks

We discussed how broken dependencies may be protected by locks, implying that the data on which the dependency exists is only supposed to be read or written to by one thread at a time §6.3.3. This makes it easy to reason that a reordering of the dependency would not be observable by other threads at runtime.

Data/Address Dependencies Protected by Control Dependencies

If data or address dependencies happen within the scope of a control dependency, as long as the control dependency is being preserved, there is no risk of reordering being observed by another thread, even if the data/address dependency is broken.

Broken Dependencies Running into Unknown Code

Both the MEndChecker and the BEndChecker face the limitations of static analysis. If an unknown component, e.g. inline assembly, gets introduced on a broken dependency chain, as a precaution, both will have to mark the dependency as preserved, since they cannot confidently reason about the unknown component, allowing the broken dependency to go by unnoticed.

The BEndChecker especially faces a significant amount of inline assembly. And we believe this is one of the main reasons it was not able to find any more broken dependency orderings.

6.3.5. Limits of the Algorithm

Syntactic dependencies are a fundamental problem reflecting the limits of static analysis. Given our goal of using such a tool in a production environment and the ensuing need to reduce false positives, this marks a hard limitation in reaching that goal. This means a balance needs to be made; we increase the number of false negatives, i.e. dependency orderings which are deliberately not annotated out of fear of reporting a false positive, in favor of reducing the amount of false positives. The problem is that even that is an idealized scenario, as a dependency ordering existing can be context-dependent. Consider the Linux kernel's implementation of a circular doubly-linked list in include/linux/list.h. In the function list_splice_tail_init(), which joins two lists, there is a control dependency, as depicted in Listing 6.10 where two function calls have been inlined for easier reading. Both writes are part of the same function call. That control dependency is only a semantic dependency if the lists that are being joined are accessed by at least two threads. If that were not the case and only one thread would be accessing the list, the dependency becomes syntactic. It becomes syntactic because there would be no other thread that would be able to observe anything happening out of order, and even if a compiler were to break the dependency, as we discussed in §2.1, the CPU would still ensure that instructions appear to the executing thread in program order.

If there are multiple threads accessing the list, the function arguments passed to the list_del() function will each be heading a dependency, starting at the READ_ONCE() that the returned the pointers before the function call. That READ_ONCE() is necessary to order the dependent calls along the deletion of the list entry and to avoid unintended behavior in the presence of more than one thread.

6.4. Assumptions and Overestimations

To ensure the functionality of the StatDepChecker and to reduce the number of false positives, we make several assumptions and overestimations:

Untrackable dependency chains As soon as the StatDepChecker discovers that a dependency chain is moving out of the realms of its capabilities it will throw it away, causing any dependency on that dependency chain not to be annotated in the first place, or if the dependency originates at an annotated dependency beginning, it will mark the dependency with that particular ID as preserved. In both cases, the StatDepChecker is avoiding false positives.

Volatile accesses in IR map to marked accesses in C source code We make the assumption that all loads and stores marked volatile in LLVM IR are the result of a marked access in C source code since marked accesses are implemented as volatile casts [Ling]. Strictly speaking, this will also include accesses to bare volatile variables in C. An overestimation we are willing to make, as such variables can still carry dependency chains and such accesses do exist in the kernel.

6.5. Performance

We conduct our testing on an AMD EPYC 7713P system with 64 cores, 128 threads, and 512 Gigabytes of RAM, running NixOS 23.05. We consider the overhead of the current version of the StatDepChecker negligible. With the annotator and the MEndChecker enabled such a build takes 2:56 minutes on our system and 2:55 minutes with StatDepChecker disabled.

6.6. Conclusion

The StatDepChecker proves the point that the LKMM's dependency orderings are sometimes being broken by compiler optimizations. Yes, all of the true positives we described above were protected by locks, but there is no guarantee that such breakages will always be protected by locks. However, the StatDepChecker is constrained by the limits of static analysis in the dependency chains it sees when annotating and the dependency chains it can see when checking. In the first case, dependencies may get missed if an unknown component exists on the dependency chain, and in the second case, dependency chains may have to be marked preserved as a precaution since the StatDepChecker cannot confidently reason otherwise. We believe that this is the main reason for the low runtime overhead, but we would like to increase dependency coverage. Especially since we have seen in §6.3.3 that a broken dependency does not get annotated because of an unknown component on the dependency chain. One way to increase dependency coverage is to add control and data dependency support for which we believe we already have a solid foundation. But there is another way - the dynamic dependency checker - which we will discuss in the following chapter.

```
static int ksm_scan_thread(void *nothing)
{
        // [...]
        while (!kthread_should_stop()) {
                mutex_lock(&ksm_thread_mutex); // Lock
                [...]
                if (ksmd_should_run())
                        ksm_do_scan(ksm_thread_pages_to_scan);
        // [...]
}

// [...]
static void ksm_do_scan(unsigned int scan_npages)
{
        // [...]
        while (scan_npages-- && likely(!freezing(current))) {
                // [...]
                cmp_and_merge_page(page, rmap_item);
                // [...]
        }
}

// [...]
static void cmp_and_merge_page(struct page *page, struct ksm_rmap_item *rmap_item)
{
        // [...]
        // Dependency beginning
        stable_node = page_stable_node(page);
        if (stable_node) {
                if (stable_node->head != &migrate_nodes &&
                    get_kpfn_nid(READ_ONCE(stable_node->kpfn)) !=
                    NUMA(stable_node->nid)) {
                        stable_node_dup_del(stable_node);
                        stable_node->head = &migrate_nodes;
                        // Dependency ending via "stable_node->head"
                        list_add(&stable_node->list, stable_node->head);
                }
        }
        // [...]
}
```

Listing 6.3: A broken dependency ordering in mm/ksm.c::2114 [Linh].

```
static int nfs_server_return_marked_delegations(struct nfs_server *server,
        void __always_unused *data) {}
{
        // [...]
        rcu_read_lock(); // Lock
        if (place_holder)
                // Dependency beginning
                delegation = rcu_dereference(NFS_I(place_holder)->delegation);
        if (!delegation || delegation != place_holder_deleg)
                delegation = list_entry_rcu(server->delegations.next,
                                            struct nfs_delegation, super_list);
        // Dependency ending via "delegation"
        list_for_each_entry_from_rcu(delegation, &server->delegations, super_list) {
        // [...]
}
```

Listing 6.4: A broken dependency ordering in fs/nfs/delegation.c::613 [Linc].

```
vm_fault_t do_swap_page(struct vm_fault *vmf)
{
        // [...]
        locked = folio_lock_or_retry(folio, vma->vm_mm, vmf->flags); // Lock
        // [...]
        if (swapcache) {
                // [...]
                // Dependency beginning
                folio = page_folio(page);

                // [...]
                // Dependency ending via "folio"
                if ((vmf->flags & FAULT_FLAG_WRITE) && folio == swapcache &&
                    !folio_test_ksm(folio) && !folio_test_lru(folio))
                        lru_add_drain();
        // [...]
}
```

Listing 6.5: A broken address dependency in mm/memory.c::3878 [Lini].

```c
static int __stm_source_link_drop(struct stm_source_device *src,
                struct stm_device *stm)
{
        // [...]
        spin_lock(&stm->link_lock); // Lock
        spin_lock(&src->link_lock); // Lock
        // Dependency beginning
        link = srcu_dereference_check(src->link, &stm_source_srcu, 1);

        // [...]
        if (link != stm) {
                ret = -EAGAIN;
                goto unlock;
        }

        stm_output_free(link, &src->output);
        list_del_init(&src->link_entry);
        // Dependency ending
        pm_runtime_mark_last_busy(&link->dev);
        // [...]
}
```

Listing 6.6: A broken address dependency in drivers/hwtracing/stm/core.c::1071 [Lin14].

```
void lockdep_unregister_key(struct lock_class_key *key)
{
        [...]
        lockdep_lock(); // Lock

        // Dependency beginning in "k"
        hlist_for_each_entry_rcu(k, hash_head, hash_entry) {
                if (k == key) {
                        // Dependency ending
                        hlist_del_rcu(&k->hash_entry);
                        found = true;
                        break;
                }
        }
        // [...]
}
```

Listing 6.7: A broken address dependency in kernel/locking/lockdep.c::6455 [Line].

```c
if (test_and_set_bit(PVCALLS_FLAG_ACCEPT_INFLIGHT,
            (void *)&map->passive.flags)) {
        // Dependency beginning
        req_id = READ_ONCE(map->passive.inflight_req_id);
        if (req_id != PVCALLS_INVALID_ID &&
            READ_ONCE(bedata->rsp[req_id].req_id) == req_id) {
                map2 = map->passive.accept_map;
                goto received;
        }
        [...]
}

// [...]

// "req_id" gets overwritten with a independent value
ret = get_request(bedata, &req_id);

// [...]

//"req_id" is not part of the dependency chain anymore
if (wait_event_interruptible(bedata->inflight_req,
        READ_ONCE(bedata->rsp[req_id].req_id) == req_id)) {
        pvcalls_exit_sock(sock);
        return -EINTR;
}
```

Listing 6.8: A dependency chain value gets overwritten in a function call in drivers/xen/pvcalls_front.c::796 [Lin17].

```c
for (;;) {
        struct fib6_node *next;

        dir = addr_bit_set(args->addr, fn->fn_bit);

        next = dir ? rcu_dereference(fn->right) :
                     rcu_dereference(fn->left); // Two dependencies begin

        if (next) {
                fn = next;
                continue;
        }
        break;
}

while (fn) {
        // [...]
        fn = rcu_dereference(fn->parent); // two dependencies end
}
```

Listing 6.9: An unintentional annotation of a dependency that requires at least one loop iteration in net/ipv6/ip6_fib.c::1563 [Linj].

```c
if (!(READ_ONCE(list->next) == head)) {  // <- list_empty(list)
        __list_splice(list, head, head->next);
        WRITE_ONCE(list->next, list);  // <- INIT_LIST_HEAD(list)
        WRITE_ONCE(list->prev, list);  // <- INIT_LIST_HEAD(list) cont.
}
```

Listing 6.10: A control dependency in the kernel's circular doubly-linked list implementation [Lind].

7. The Dynamic Dependency Checker

We present the Dynamic Dependency Checker (DynDepChecker), designed to take over when the StatDepChecker reaches its limits. Unlike the StatDepChecker, the DynDepChecker operates at runtime, allowing it to view code as it is being executed. At this point, all unknown components within the code are resolved, e.g., function pointers, making it possible for the DynDepChecker to see dependency chains the StatDepChecker cannot. This naturally extends the design of the StatDepChecker, as shown in Figure 7.1, where instead of reporting breakages it simply delegates its checks to the DynDepChecker when it is unsure.

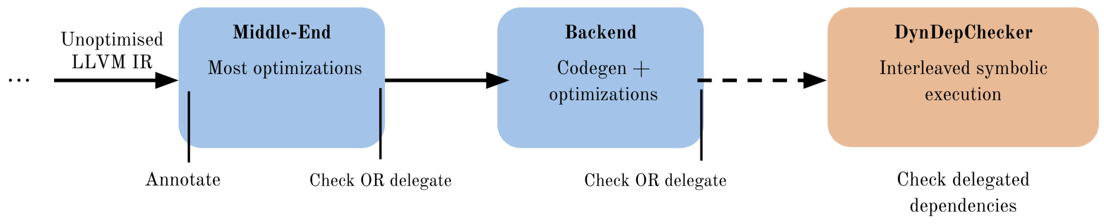


Figure 7.1.: The DynDepChecker extends the StatDepChecker

7.1. Design

At the core of our design for the DynDepChecker lies the idea of repurposing symbolic execution for data flow tracking. As presented in §4.5, symbolic execution was originally intended to achieve full program coverage for software testing. Instead, we aim to use symbolic execution for tracking dependency chains. By marking only the beginning of a dependency ordering's dependency chain as symbolic, we assume that seeing a symbolic value at the end of the dependency implies a preserved dependency chain. If we see a concrete value at the dependency ending instead, the dependency chain must have been broken as the symbolic value could be propagated until the end of the dependency chain.

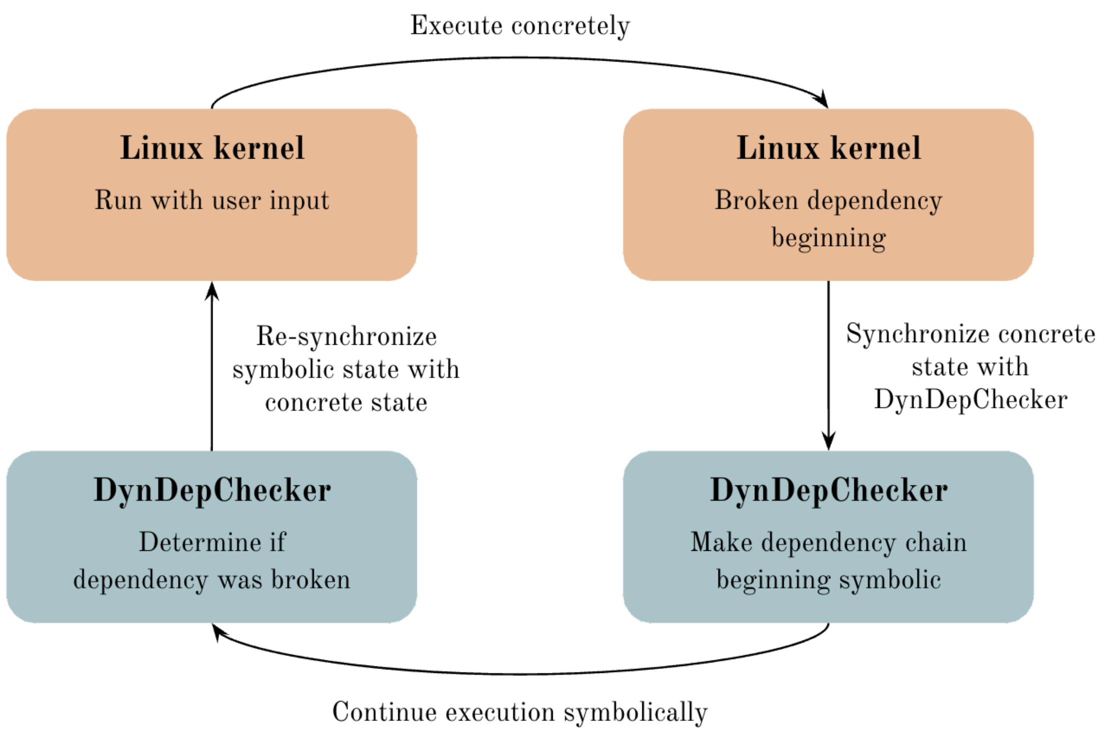

Execute concretely
Linux kernel
Run with user input
Linux kernel
Broken dependency beginning
Re-synchronize symbolic state with concrete state
Synchronize concrete state with DynDepChecker
DynDepChecker
Determine if dependency was broken
DynDepChecker
Make dependency chain beginning symbolic
Continue execution symbolically

Figure 7.2.: The dynamic dependency checker.

Choosing a Framework

Implementing such a design requires a framework with support for a mechanism to only make a part of a program symbolic. For this, we choose angr as it supports interleaved symbolic execution via Symbion, provides easy programmability, and it is still actively maintained. Both, angr and Symbion, are discussed further in §4.6 and §4.6.1 respectively. Symbion is technically not being maintained anymore, but one of its creators as well as the angr community were still very happy to help with any questions we had. Symbion provides the additional benefit of being able to switch between a concrete execution and a (partially) symbolic execution. That way the overhead of the symbolic execution can be kept to a minimum as the partial symbolic execution will only happen along the dependency chain. The dependency chain will be reached with a concrete execution, keeping the overhead to a minimum. Additionally, Symbion decouples the symbolic execution and concrete execution by having the concrete execution run on a potentially remote host. Symbion merely requires that the concrete target it connects to is running a GDB server.

7.2. Implementation

Based on the second idea, we implement a proof of concept which we have validated for the simple StatDepChecker case shown in Listings 6.1 and 6.2. Our proof of concept runs on X86 only, as Symbion does not support arm64 out of the box. We discuss the PoC further in §7.3. Our implementation can be viewed at: `https://github.com/PBHDK/mthesis-angr/tree/main/dyndepchecker`. Our corresponding angr-dev repository, which the angr developers maintain for managing an angr repository, is, is available at: `https://github.com/PBHDK/mthesis-angr-dev` The DynDepChecker PoC can be partitioned into the following flow of events.

7.2.1. Delegating Dependency Checks to the DynDepChecker

During kernel compilation and when dependency delegations are enabled, whenever the StatDepChecker encounters a point in a dependency chain where it cannot confidently tell if the dependency chain got broken or not, instead of marking the dependency as preserved out of fear of false positives, it can instead delegate the dependency to the DynDepChecker. For example, if a dependency chain runs into an intrinsic function that returns a value, there is the possibility that the value being returned should be part of the dependency chain. Since the MEndChecker and the BEndChecker cannot traverse intrinsic functions because they are implemented as inline assembly, they cannot confidently determine whether the return value is part of the dependency chain. If they add the return value to the dependency chain, although it is not part of it, they taint their analysis by adding wrong values to the dependency chain and potentially reporting a dependency as preserved, although it was not. If they do not add the return value to the dependency chain, although it is part of it, they taint their analysis by missing links in a dependency chain and potentially reporting a dependency as broken even though it was preserved. To get a definitive answer, they can delegate the dependency checks to the DynDepChecker, for which inline assembly does not pose a problem. This is done by annotating the dependency with LLVM PC section metadata, which will be visible to the DynDepChecker via the compiled binary. Users will end up with a Linux kernel binary that contains the dependency delegations made by the StatDepChecker as well as a corresponding Linux kernel image. We implement the delegation mechanism for the MEndChecker.

7.2.2. Starting the Concrete Target

Users start the concrete target by booting up a qemu virtual machine with the previously generated Linux kernel image and instruct qemu to start a GDB server alongside that will prevent any execution until a remote connects to it.

7.2.3. Starting the DynDepChecker

Once the qemu VM is waiting for an incoming connection to the GDB server, users can start the DynDepChecker by invoking the corresponding Python script. The DynDepChecker will look at the previously generated Linux kernel binary, parse the PC sections, and then connect to the GDB server of the qemu VM.

7.2.4. Performing Dependency Checks at Runtime

Once connected, the DynDepChecker begins its analysis. Having parsed the PC sections before, it knows at which addresses all dependency beginnings lie. Using angr, it can set breakpoints at all dependency beginnings. The concrete execution will proceed remotely until one of the breakpoints is reached. When a breakpoint is reached, execution is transferred to the DynDepChecker and Symbion. The DynDepChecker inspects the disassembly of the current basic block, marks the register into which the dependency reads as symbolic, and proceeds with the symbolic execution until it reaches the corresponding dependency ending instruction. Upon reaching the instruction, it determines whether the corresponding operand is symbolic and reports a dependency breakage to the user if necessary. Afterward, it transfers the execution back to the concrete target. Where angr will ensure that the concrete state is updated to match the progress made by the symbolic execution.

7.3. Evaluation

We find ourselves in a conundrum. The success of the DynDepChecker hinges on the capabilities of the StatDepChecker's Annotator. Even a perfect DynDepChecker has to work with the annotations it receives from the StatDepChecker, meaning that the DynDepChecker only strengthens the StatDepChecker's checking mechanisms, not its annotation mechanisms. Whilst our implementation of the DynDepChecker proves what we deem an interesting design, it does not provide the leverage we hoped for in terms of checking broken dependencies. The DynDepChecker works in cases where a dependency chain is fully visible in unoptimized IR but becomes partially invisible due to compiler optimizations, e.g. because inline assembly is introduced on the dependency chain. This can happen, but the true problem is dependency chains being already partially invisible at compile time, such as those in Listing 6.4.

7.3.1. Coverage

Our PoC implementation is very limited in what code regions it can reach as we have only tested it for a full boot of a default-config kernel. However, as we cannot make any assumptions about where dependency annotations may lie within the codebase, it would be important to gain as much coverage as possible for the DynDepChecker. We envision a design where we can connect the concrete target to a coverage-guided fuzzer such as syzkaller, which we described in §4.8.1. syzkaller is designed to achieve maximum coverage for a given Linux kernel and could prove valuable in reaching the annotated dependencies. syzkaller's convergence to annotated dependencies may even be improved with a mechanism where the StatDepChecker can guide syzkaller through the information it gathered during annotating and checking. A simple idea we had for such a mechanism was to selectively add the compiler instrumentation on which syzkaller relies along the dependency chain of a given dependency. That way, achieving code coverage in the kernel would equal achieving dependency chain coverage from syzkaller's point of view, and any input that would not get within reach of a dependency chain would cause syzkaller to "run dark."

7.4. Performance

For our testing purposes, it suffices to have the DynDepChecker and the concrete target run on the same machine. We reuse the same machine we used for testing the StatDepChecker, outlined in §6.5.

We measure the following runtimes:

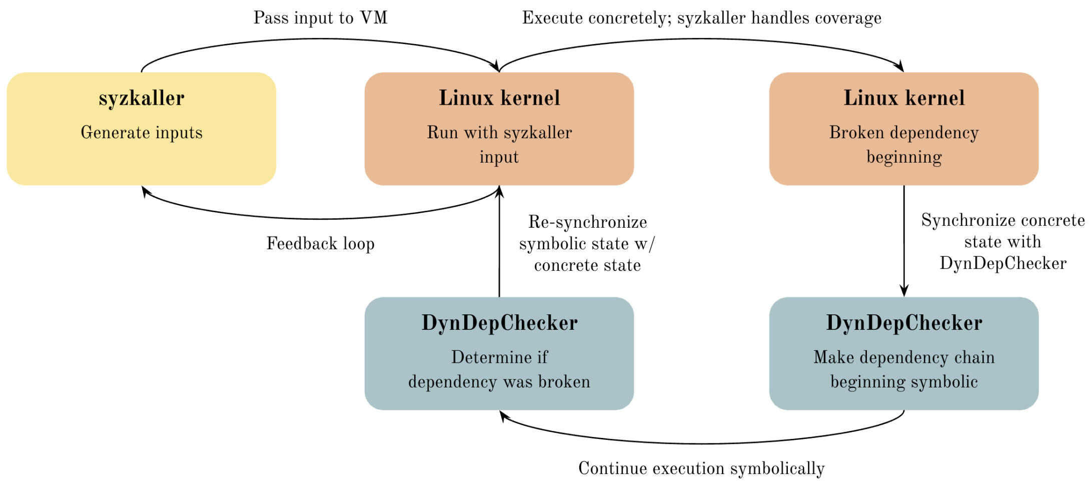

Pass input to VM
Execute concretely; syzkaller handles coverage
syzkaller
Generate inputs
Linux kernel
Run with syzkaller input
Linux kernel
Broken dependency beginning
Feedback loop
Re-synchronize symbolic state w/ concrete state
Synchronize concrete state with DynDepChecker
DynDepChecker
Determine if dependency was broken
DynDepChecker
Make dependency chain beginning symbolic
Continue execution symbolically

Figure 7.3.: A coverage-guided fuzzer such as syzkaller could help us scale and reach the dependency annotations at runtime.

DynDepChecker startup We measure the time for starting the DynDepChecker, i.e., loading the Linux kernel binary into angr and parsing the PC sections at around 3:20 minutes.

Linux kernel boot with DynDepChecker Booting the Linux kernel while checking for our delegated test case dependency takes roughly 4 seconds. We measure the case where the dependency does not get broken, as the broken dependency reads from an unknown address and, therefore, prevents the boot process from continuing after it has been checked. Both cases should have a similar compute overhead, so we still deem this number representative.

Linux kernel boot without DynDepChecker Booting the same Linux kernel image without involving the DynDepChecker takes roughly 2.8 seconds.

Loading the Linux kernel binary into the DynDepChecker, i.e., angr therefore marks the main overhead. However, for every analysis run, this overhead should be constant. The analysis itself marks an overhead of roughly 30 percent, which sounds significant in relative terms but only adds a mere 1.2 seconds in absolute terms.

7.5. An Alternative Design

Our first design idea for the DynDepChecker was not based on symbolic execution but instead tried to leverage the Linux Kernel Memory Sanitizer (KMSan), which we described in §4.4.1. What KMSan does at runtime is not unlike what the StatDepChecker does at compile time. It works like somewhat of an inversion of the StatDepChecker. For instance, consider Listing 7.1, it is KMSan's job to determine that using `a_dep` in the conditional is a bug, as it is uninitialized. To do so, it would have to connect the use of `a_dep` in the conditional to the declaration of `a_dep` earlier, using something similar to a dependency chain, except that connecting two instructions is a sign of something being wrong. In contrast, with the StatDepChecker, it is a sign of something being right. KMSan refers to this as value poisoning.

```c
/* Not initialised */
int a;
int a_dep;

/* OK per C standard. */
a_dep = a;

/* Not OK! Bug! */
if(a_dep)
        /* Branch */
```

Listing 7.1: Using an uninitialized value in an assignment does not mark undefined behavior; using it in a conditional, on the other hand, does [Ale21].

Our first idea for the DynDepChecker was to invert KMSan's value poisoning mechanism and use it for dependency chain tracking. The correspondence would look somewhat like in Listing 7.2. We abandoned this idea as KMSan relies on a significant amount of compiler instrumentation to implement the value poisoning. That makes it susceptible to the same problems that StatDepChecker faces, e.g., inline assembly.

7.6. Conclusion

Our implementation of the dynamic dependency checker proves that broken dependencies can be checked by repurposing symbolic execution. By connecting it to the StatDepChecker, we have demonstrated that the DynDepChecker could support the StatDepChecker by checking dependency orderings, where unknown code, such as inline assembly or intrinsics, was introduced through compiler optimizations. However, the DynDepChecker can only make those checks if the unknown components did not exist when the Annotator ran, meaning that the DynDepChecker is limited to

```
/* Allocation -> READ_ONCE() */
int a;
int a_dep;

/* Poisoning -> Dependency chain */
a_dep = a;

/* Use of poisoned value (in annotated inst) -> WRITE_ONCE() */
if(a_dep)
        /* Do something */
```

Listing 7.2: How Listing 7.1 relates to dependency chains.

the dependencies the Annotator saw when it performed its static analysis. Given its external dependencies on angr and unresolved challenges regarding coverage, we do not see it being used in production but as a foundation for future research instead.

Part IV.

Conclusion

8. Discussion and Future Directions

Previous sections have alluded to several items of future work. In the following, we will summarize, expand on, and add to those items and conclude this thesis.

8.1. Future Work

The overarching piece of future work for us is to mold the results of this thesis and any future work that follows into a scientific paper. We outline improvements to the individual parts of the DepChecker below.

8.1.1. StatDepChecker

Upstreaming Our goal for the StatDepChecker is to upstream it into LLVM for experimental use at first. We did not manage to submit an LLVM RFC in time but aim to do so soon after submitting this thesis.

RCU StatDepChecker Private discussions with Paul E. McKenney centered on a specialization of the StatDepChecker for the Linux kernel's RCU implementation. The central idea is that RCU loads such as `rcu_dereference()` always head a dependency chain. That means that any dependency chain headed by an `rcu_dereference()` always carries a semantic dependency. The dependency chain may be terminated when running into a `rcu_pointer_handoff()` or out of the outermost RCU read-side critical section. Given the RCU-related kernel configuration options as well as the RCU API, future work might consist of having the dependency checker treat RCU dependency orderings differently than those starting at regular marked reads.

Implementation Improvements

As of now, our implementation still faces some limitations.

Pointer aliasing The current implementation of the pointer aliasing approximation in dependency chains only models two levels: the pointee level and the pointer level. Dependency chain values may switch between the two, but once a value leaves the pointer level, the StatDepChecker stops tracking the dependency chain to avoid false positives.

Data dependency support Adding data dependency should be straightforward and should only require further testing.

Control dependency support We have explored several implementations for adding control dependency support, which we further discuss in §6.2. A union of our explored approaches will hopefully yield a satisfactory control dependency implementation for the StatDepChecker.

Differentiate between full and partial address/data dependencies To detect full-to-partial conversions such as the one shown in Listing 5.2, the StatDepChecker would have to be able to differentiate between full and partial address/data dependencies. We have explored an approach that tracks two sets of dependent values, as explained in §6.2, but had initially prioritized to not further pursue it. Once an initial version of the StatDepChecker has been upstreamed, we would like to continue our work on this approach.

Relaxed mode A READ_ONCE() can also carry dependencies into unmarked accesses, i.e., its dependency chain. Such implicit dependencies do not require the existence of a second dependent READ_ONCE() or WRITE_ONCE() [Ling]. We have started work on a "relaxed mode" for the StatDepChecker, which would only (or additionally) annotate such implicit dependencies and check them for breakages. To avoid an unnecessary annotation overhead, not all dependency chains would be annotated, but only those that do not run into a second marked access. However, we were not able to complete it within the scope of this thesis.

Link-time optimization Link-time optimization (LTO) promises more extensive compiler optimizations across translation units. We have begun work on running the StatDepChecker as part of the LLVM ThinLTO and LTO pipelines but were unable to complete our implementation within the scope of this thesis.

Control-flow graph pruning Currently, the StatDepChecker is not aware of which code regions it has seen before. That means that functions that are called more than once will be traversed more than once. In most cases, this will be required as the StatDepChecker will be entering the functions in different contexts, i.e., there will be different dependency chains running in and out of a given function. However, all dependencies that solely occur within that function need only to be tracked once and are currently re-tracked every time the StatDepChecker enters that function. It will not make duplicate annotations, but we believe that the computational overhead would be worth looking into.

Loops Our current implementation does not handle loops. By not traversing loops in the first place and, therefore, not annotating dependencies that require a full loop iteration in order to exist, we appear to mitigate the extent of the StatDepChecker not handling loops. However, we have encountered a case where a dependency does not require a loop iteration to exist in unoptimized IR but does require a loop iteration in optimized IR. Since the StatDepChecker does not traverse loops, this leads to a false positive, and being able to traverse loops, even just once or twice, would mitigate this false positive. This case is further discussed in §6.3.3.

Suggest fixes to users Suggesting fixes to users would pose a valuable feature, however, we are unsure how to implement it. Suggesting a fix would imply knowing where exactly a dependency chain got broken. However, given that dependency chains usually split up along the different control flow paths, users would face several "loose ends" of dependency chains. However, one might settle for simply printing the dependency chain to users if desired, leaving it to users to determine what values are missing and where in the code the breakage occurs. We had previously implemented such a feature but did not find it useful for our testing purposes, as looking at the source code usually made it obvious where a breakage occurred.

Produce a minimal working example of a broken dependency A step towards a solution for the above point could be producing a minimal working example of a broken dependency. That would still not narrow the report down to the exact point of breakage, but it would most likely make the point of breakage more obvious. This may be achieved with a tool such as C-Reduce [Reg+12].

Pinpoint the optimization pass which caused a dependency breakage To help inform compiler engineers as to where in the optimization pipeline dependency breakages occur, at the expense of a significant performance overhead, we could pinpoint the optimization pass that is performing a dependency-breaking optimization by having the StatDepChecker run its checker pass after every LLVM optimization pass. This would cause a runtime overhead by the factor of the number of optimization passes.

Individual handling of intrinsics To achieve broader coverage of dependency chains running through compiler intrinsics, it might be interesting to handle intrinsic functions [LLVb] individually by deeming certain intrinsic functions dependency-preserving.

8.1.2. DynDepChecker

Concretization of symbolic addresses angr offers different concretization strategies for deciding what happens when a program reads from a symbolic address. In our current implementation, the SYMBOLIC_WRITE_ADDRESSES flag is passed to angr, which allows fully symbolic writes at the expense of performance. Investigating different concretization strategies and potentially avoiding them altogether could lead to a speedup of the analysis.

Improving coverage As discussed in §7.3.1, we could see the DynDepChecker being extended to receive inputs from a coverage-guided fuzzer such as syzkaller to improve dependency coverage.

Scaling out To take advantage of Symbion being able to decouple the concrete target from the symbolic execution as well as the idea of connecting a coverage-guided fuzzer to the concrete target, we also envision a design where several concrete targets run in parallel, all of which are guided by one syzkaller instance. In addition to the points above, parallelizing the search for dependency annotations in the kernel would hopefully provide a significant increase in speed.

Explore different states when returning to concrete execution Our current implementation of the PoC would theoretically restart the kernel for every dependency it needs to check. Since we only tested the PoC with one delegated dependency, this is not a problem. If more dependencies were involved, we would like to explore different approaches when returning to the concrete execution, especially if we could connect the DynDepChecker to a coverage-guided fuzzer. After checking a dependency, one could either restart the execution with the same or a different input, return to the starting point of the dependency chain, or continue at the point where the dependency chain ended. We believe that the latter would hold the risk of missing dependencies that started on the dependency chain that was just checked.

arm64 Support Our PoC only runs on X86, as Symbion does not support arm64 as of right now. Discussions with one of the Symbion creators have led us to believe that arm64 support for Symbion may not be as big of a hurdle as it sounds. This future work would present a valuable open-source contribution besides the obvious benefit of having the DynDepChecker run on an architecture where dependency breakages matter, as they do not on X86, which does not reorder writes/reads ahead of previous reads [OSS09].

8.2. Conclusion

This work investigated whether dependency orderings in the Linux kernel get broken by optimizing compilers and, if so, to what extent this occurs. As so often, the results present themselves in a nuanced way. To our knowledge, our work was the first to prove that LKMM dependency orderings get broken by compilers. Our implementation of a dependency checker only rarely reported dependency breakages. However, this does not necessarily mean that they only occur rarely, as it may result from the caution the DepChecker has to exercise to avoid false positive reports. The current version of the DepChecker is not even annotating one of its trophies anymore since the dependency chain runs through an intrinsic function in unoptimized LLVM IR, which the DepChecker cannot confidently reason about. Dependency breakages keep themselves well hidden from static analysis. So, we explored a more potent checking mechanism: the DynDepChecker. Still, all efforts in this direction remain constrained by the concessions the StatDepChecker's Annotator has to make to avoid false positive annotations. We outlined future work for improving our current implementation in §8, but we believe that the most effective way forward would be a mechanism in the Linux kernel allowing users to mark the dependency chains. This could be implemented through the type system, for example, as suggested by Garg et al. in the proposal for a `_dependent_ptr` type qualifier in C [GMR19]. This would hopefully alleviate most of the hurdles static analysis poses and yield a more potent DepChecker.

A. Appendix

Below, we will links to the current implementations of our work. All of these should still be considered work-in-progress. We plan to continue working on them after submitting this thesis and plan on moving them to a more centralized location. For now, our implementation of the StatDepChecker and the DynDepChecker have been moved to static mirrors of the development trees so the links will not become outdated.

A.1. Running the StatDepChecker

Our implementation of the Annotator and MEndChecker is available at `https://github.com/PBHDK/mthesis-llvm-project/blob/main/llvm/lib/Transforms/Utils/LKMMDependenceAnalysis.cpp`, and Martin Fink's implementation of the BEndChecker is available at `https://github.com/martin-fink/llvm-project/tree/dep-checker-backend`. We provide a shell.nix file for loading project dependencies and a build script. From the root of our LLVM repository, obtain a custom version of clang, which includes the Annotator as well as the MEndChecker with the following commands:

```
$ ./proj_bdo_llvm_build.py config
$ ./proj_bdo_llvm_build.py build clang
```

Once the build has succeeded, move to our fork of the Linux kernel, which is available at `https://github.com/PBHDK/mthesis-linux`. We again provide a shell.nix file, as well as a build script. The shell.nix should automatically use the clang binary that includes the StatDepChecker, provided that the correct path to the LLVM build directory is specified within the shell.nix. A default-configuration Linux kernel for arm64 with dependency checks enabled can be built with the following commands:

```
$ ./proj_bdo/linux_build.py config defconfig arm64
$ ./proj_bdo/linux_build.py build fast arm64
```

This should yield a Linux kernel binary with one delegated dependency for the DynDepChecker. The StatDepChecker output will be written to a file called build_output.err.

A.2. Running the DynDepChecker

The angr developers provide the angr-dev repository for managing an angr development environment. We provide the following forks:

1. angr: `https://github.com/PBHDK/mthesis-angr`
2. angr-dev: `https://github.com/PBHDK/mthesis-angr-dev`
3. buildroot: `https://github.com/PBHDK/mthesis-buildroot`

Again, we provide shell.nix files, in this case, based on the ones that angr already provides. Assuming that qemu is installed on the system, a buildroot image has been generated, and invoking the DynDepChecker involves the following commands.

First, start a qemu instance in a separate shell:

```
$ qemu-system-x86_64 \
  -m 2G \
  -smp 2 \
  -kernel linux/arch/x86/boot/bzImage \
  -append "console=ttyS0␣root=/dev/sda1␣earlyprintk=serial␣net.ifnames=0␣nokaslr" \
  -drive file=buildroot/output/images/disk.img,format=raw \
  -net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:10021-:22 \
  -net nic,model=e1000 \
  -enable-kvm \
  -nographic \
  -pidfile vm.pid \
  2>&1 -monitor none -s -S| tee vm.log
```

The command starts a GDB server and will not progress until a client connects to that GDB server. The above command is based on the syzkaller documentation [syzb]. From within the angr repository, start the DynDepChecker PoC with:

```
$ python dyndepchecker/ddd_x86.py
```

The DynDepChecker will connect to the GDB server of the qemu VM, search for a delegated dependency, and, if found, terminate after analyzing it.

List of Figures

List of Listings

Bibliography

[AF94] Hagit Attiya and Roy Friedman. "Programming DEC-Alpha Based Multiprocessors the Easy Way (Extended Abstract)." In: *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '94. New York, NY, USA: Association for Computing Machinery, Aug. 1994, pp. 157–166. ISBN: 978-0-89791-671-4. DOI: `10.1145/181014.192323`. (Visited on 09/15/2023).

[AG96] Sarita V. Adve and Kourosh Gharachorloo. "Shared Memory Consistency Models: A Tutorial." In: *Computer* 29.12 (Dec. 1996), pp. 66–76. ISSN: 0018-9162. DOI: `10.1109/2.546611`.

[Ale21] Alexander Potapenko. *KernelMemorySanitizer - a Look under the Hood*. FaMAF-UNC, Córdoba, Argentina, July 2021. URL: `https://www.youtube.com/watch?v=LNs2U-3m3yg` (visited on 10/16/2023).

[Alg+12] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. "Fences in Weak Memory Models (Extended Version)." In: *Formal Methods in System Design* 40.2 (Apr. 2012), pp. 170–205. ISSN: 1572-8102. DOI: `10.1007/s10703-011-0135-z`.

[Alg+18] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan 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*. Williamsburg VA USA: ACM, Mar. 2018, pp. 405–418. ISBN: 978-1-4503-4911-6. DOI: `10.1145/3173162.3177156`.

[Alg+19] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. *Who's Afraid of a Big Bad Optimizing Compiler?* July 2019. URL: `https://lwn.net/Articles/793253/` (visited on 10/27/2023).

[Alg+21] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. "Armed Cats: Formal Concurrency Modelling at Arm." In:

ACM Transactions on Programming Languages and Systems 43.2 (June 2021), pp. 1–54. ISSN: 0164-0925, 1558-4593. DOI: `10.1145/3458926`.

[Alg10] Jade Alglave. "A Shared Memory Poetics." PhD thesis. Paris, Nov. 2010. URL: `http://www0.cs.ucl.ac.uk/staff/j.alglave/these.pdf`.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. *Herding Cats - Modelling, Simulation, Testing, and Data-Mining for Weak Memory*. Jan. 2014. arXiv: `1308.6810 [cs]`. URL: `http://arxiv.org/abs/1308.6810`.

[Arm09] Arm Ltd. *Barrier Litmus Tests and Cookbook*. Nov. 2009. URL: `https://developer.arm.com/documentation/genc007826/latest` (visited on 10/16/2023).

[Att+93] Hagit Attiya, Soma Chaudhuri, Roy Friedman, and Jennifer L. Welch. "Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies." In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '93. New York, NY, USA: Association for Computing Machinery, Aug. 1993, pp. 241–250. ISBN: 978-0-89791-599-1. DOI: `10.1145/165231.165263`.

[Bat14] Mark John Batty. "The C11 and C++11 Concurrency Model." PhD thesis. University of Cambridge, Nov. 2014. URL: `https://www.cs.kent.ac.uk/people/staff/mjb211/docs/toc.pdf`.

[BP09] Gérard Boudol and Gustavo Petri. "Relaxed Memory Models: An Operational Approach." In: *ACM SIGPLAN Notices* 44.1 (Jan. 2009), pp. 392–403. ISSN: 0362-1340, 1558-1160. DOI: `10.1145/1594834.1480930`.

[Cla] The ClangBuiltLinux Maintainers. *ClangBuiltLinux*. Source Code. URL: `https://github.com/ClangBuiltLinux` (visited on 10/27/2023).

[cpp] The cppreference Authors. *Atomic Types - Cppreference.Com*. Documentation. URL: `https://en.cppreference.com/w/c/language/atomic` (visited on 10/16/2023).

[CT12] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. 2. ed. Amsterdam Heidelberg: Elsevier, Morgan Kaufmann, 2012. ISBN: 978-0-08-091661-3.

[Dea20] Will Deacon. *Dependency Ordering in the Linux Kernel*. Aug. 2020. URL: `https://lpc.events/event/7/contributions/821/` (visited on 10/14/2023).

[Dea21] Will Deacon. *The Never-Ending Saga of... Control Dependencies*. 2021. URL: `https://lpc.events/event/11/contributions/973/` (visited on 10/14/2023).

[DSM18] Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. "Bounding Data Races in Space and Time." In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 242–255. ISBN: 978-1-4503-5698-5. DOI: `10.1145/3192366.3192421`.

[EH22] Marco Elver and Paul Heidekrüger. *Status Report: Broken Dependency Orderings in the Linux Kernel*. Dublin, Ireland, Sept. 2022. URL: `https://lpc.events/event/16/contributions/1174/` (visited on 10/14/2023).

[Ewi] Larry Ewing. *Tux*. URL: `https://isc.tamu.edu/~lewing/linux/` (visited on 10/27/2023).

[Fen23] Boqun Feng. *Re: Broken Address Dependency in mm/ksm.c::cmp_and_merge_page()*. Jan. 2023. URL: `https://lore.kernel.org/all/Y9GTgdMnGu6OxUZC@boqun-archlinux/` (visited on 10/27/2023).

[Fer87] Jeanne Ferrante. "The Program Dependence Graph and Its Use in Optimization." In: *ACM Transactions on Programming Languages and Systems* 9.3 (1987), p. 31.

[Fin23] Martin Fink. *Detecting Address Dependencies Broken by the LLVM Backend in the Linux Kernel*. Guided Research Report. Technical University of Munich, Apr. 2023, p. 5. URL: `https://github.com/TUM-DSE/research-work-archive/blob/main/archive/2022/winter/docs/gr_fink_dependencies-lk-backend.pdf` (visited on 10/19/2023).

[GMR19] Akshat Garg, Paul E McKenney, and Ramana Radhakrishnan. *_dependent_ptr to Simplify Carries a Dependency*. July 2019. (Visited on 10/24/2023).

[Gri+20] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. "SYMBION: Interleaving Symbolic with Concrete Execution." In: *2020 IEEE Conference on Communications and Network Security (CNS)*. June 2020, pp. 1–10. DOI: `10.1109/CNS48642.2020.9162164`.

[Hei21] Paul Heidekrüger. "Dependency Ordering in the Linux Kernel." Bachelor's Thesis. Munich: Technical University of Munich, Nov. 2021. URL: `https://dse.in.tum.de/wp-content/uploads/2022/01/DoitLK_thesis.pdf` (visited on 10/14/2023).

[Hei22a] Paul Heidekrüger. *Broken Address Dependency in mm/ksm.c::cmp_and_merge_page() - Paul Heidekrüger*. Apr. 2022. URL: `https://lore.kernel.org/all/YmKE%2FXgmRnGKrBbB@Pauls-MacBook-Pro.local/` (visited on 10/15/2023).

[Hei22b] Paul Heidekrüger. *Dangerous Addr to Ctrl Dependency Transformation in fs/nfs/delegation.c::nfs_server_return_marked_delegations()? - Paul Heidekrüger*. Apr. 2022. URL: `https://lore.kernel.org/all/Yk7%2FT8BJITwz+Og1@Pauls-MacBook-Pro.local/` (visited on 10/15/2023).

[Hei23] Paul Heidekrüger. *Re: Broken Address Dependency in Mm/Ksm.c::Cmp_and_merge_page() - Paul Heidekrüger*. Jan. 2023. URL: `https://lore.kernel.org/all/0EC00B0E-554A-4BF3-B012-ED1E36B12FD1@tum.de/#t` (visited on 10/15/2023).

[How06] David Howells. *[PATCH] Document Linux's Memory Barriers - David Howells*. Mar. 2006. URL: `https://lore.kernel.org/all/31492.1141753245@warthog.cambridge.redhat.com/` (visited on 10/14/2023).

[How14] David Howells. *[RFC][PATCH 0/5] Arch: Atomic Rework*. Feb. 2014. URL: `https://lore.kernel.org/all/21984.1391711149@warthog.procyon.org.uk/T/#u` (visited on 04/08/2023).

[KA01] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 2001. ISBN: 978-1-55860-286-1.

[Kin76] James C. King. "Symbolic Execution and Program Testing." In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: `10.1145/360248.360252`.

[KMS] The KMSan Authors. *Kernel Memory Sanitizer (KMSAN) — The Linux Kernel Documentation*. Documentation. URL: `https://docs.kernel.org/dev-tools/kmsan.html` (visited on 10/16/2023).

[Lam79] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." In: *IEEE Transactions on Computers* C-28.9 (Sept. 1979), pp. 690–691. ISSN: 1557-9956. DOI: `10.1109/TC.1979.1675439`.

[Lat02] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization." MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002. (Visited on 10/24/2023).

[Lat11] Chris Lattner. *The Architecture of Open Source Applications (Volume 1) LLVM*. Vol. 1. 2011. URL: `https://aosabook.org/en/v1/llvm.html` (visited on 09/10/2023).

[Lina] The Linux Kernel Authors. *control-dependencies.txt*. Source Code. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/tools/memory-model/Documentation/control-dependencies.txt?h=v6.5.7` (visited on 10/19/2023).

[Linb] The Linux Kernel Authors. *explanation.txt*. Documentation. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/tools/memory-model/Documentation/explanation.txt?h=v6.5.7#n468 (visited on 10/16/2023).

[Linc] The Linux Kernel Authors. *fs/nfs/delegation.c::613*. Source Code. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/nfs/delegation.c?h=v6.5.7#n613 (visited on 10/15/2023).

[Lind] The Linux Kernel Authors. *include/linux/list.h::list_splice_tail_init()*. Source Code. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/list.h?h=v6.5.7#n504 (visited on 10/19/2023).

[Line] The Linux Kernel Authors. *kernel/locking/lockdep.c::6457*. Source Code. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/locking/lockdep.c?h=v6.5.7#n6457 (visited on 10/15/2023).

[Linf] The Linux Kernel Authors. *litmus-tests.txt*. Documentation. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/tools/memory-model/Documentation/litmus-tests.txt?h=v6.5.9 (visited on 10/27/2023).

[Ling] The Linux Kernel Authors. *memory-barriers.txt*. Documentation. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/memory-barriers.txt?h=v6.1.53#n571 (visited on 09/16/2023).

[Linh] The Linux Kernel Authors. *mm/ksm.c::2114*. Source Code. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/mm/ksm.c?h=v6.5.7#n2114 (visited on 10/15/2023).

[Lini] The Linux Kernel Authors. *mm/memory.c::3878*. Source Code. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/mm/memory.c?h=v6.5.7#n3878 (visited on 10/15/2023).

[Linj] The Linux Kernel Authors. *net/ipv6/ip6_fib.c::1563*. Source Code. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/ipv6/ip6_fib.c?h=v6.5.7#n1563 (visited on 10/15/2023).

[Link] The Linux Kernel Authors. *Proper Care and Feeding of Return Values from rcu_dereference()*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/RCU/rcu_dereference.rst?h=v6.5.7 (visited on 10/15/2023).

[Lin14] The Linux Kernel Authors. *drivers/hwtracing/stm/core.c::1071*. Source Code. 2014. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/hwtracing/stm/core.c?h=v6.5.7#n1071 (visited on 10/15/2023).

[Lin17] The Linux Kernel Authors. *drivers/xen/pvcalls-front.c::796*. Source Code. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/xen/pvcalls-front.c?h=v6.5.7#n796 (visited on 10/15/2023).

[LLVa] The LLVM Authors. *Can I Compile C or C++ Code to Platform-Independent LLVM Bitcode?* Documentation. URL: https://llvm.org/docs/FAQ.html#can-i-compile-c-or-c-code-to-platform-independent-llvm-bitcode (visited on 10/16/2023).

[LLVb] The LLVM Authors. *LLVM Language Reference Manual*. Documentation. URL: https://llvm.org/docs/LangRef.html (visited on 10/15/2023).

[LLVc] The LLVM Authors. *LLVM Logo*. URL: https://llvm.org/Logo.html (visited on 10/27/2023).

[LLVd] The LLVM Authors. *LLVM PC Sections Metadata — LLVM 18.0.0git Documentation*. Documentation. URL: https://llvm.org/docs/PCSectionsMetadata.html (visited on 10/18/2023).

[LLVe] The LLVM Authors. *llvm/lib/Transform/Scalar/PartiallyInlineLibCalls.cpp::162*. Source Code. URL: https://github.com/llvm/llvm-project/blob/llvmorg-16.0.6/llvm/lib/Transforms/Scalar/PartiallyInlineLibCalls.cpp#L162 (visited on 09/11/2023).

[LLVf] The LLVM Authors. *The Clang Compiler*. URL: https://clang.llvm.org/ (visited on 09/10/2023).

[LLVg] The LLVM Authors. *The LLVM Compiler Infrastructure Project*. URL: https://llvm.org/ (visited on 09/10/2023).

[McK+17] Paul E McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Nelson Clark, Olivier Giroux, Lawrence Crowl, J. F. Bastien, and Michael Wong. *Marking Memory Order Consume Dependency Chains*. 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0462r1.pdf (visited on 10/19/2023).

[Mey19] Bertrand Meyer. *Soundness and Completeness: Defined With Precision*. Apr. 2019. URL: https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-defined-with-precision/fulltext (visited on 09/18/2023).

[MPA05] Jeremy Manson, William Pugh, and Sarita V Adve. "The Java Memory Model." In: *ACM SIGPLAN Notices*. SIGPLAN Not. 40.1 (Jan. 2005), pp. 378–391. ISSN: 0362-1340, 1558-1160. DOI: `10.1145/1047659.1040336`.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better X86 Memory Model: X86-TSO." In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 391–407. ISBN: 978-3-642-03358-2. URL: `http://link.springer.com/10.1007/978-3-642-03359-9_27`.

[PH17] David A. Patterson and John L. Hennessy. "4.5 An Overview of Pipelining." In: *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Mar. 2017. ISBN: 978-0-12-812275-4.

[Ran00] Brian Randell. "Turing Memorial Lecture Facing Up to Faults." In: *The Computer Journal* 43.2 (Jan. 2000), pp. 95–106. ISSN: 0010-4620. DOI: `10.1093/comjnl/43.2.95`.

[Reg+12] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-Case Reduction for C Compiler Bugs." In: *ACM SIGPLAN Notices* 47.6 (June 2012), pp. 335–346. ISSN: 0362-1340. DOI: `10.1145/2345156.2254104`.

[Sar+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER Multiprocessors." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. New York, NY, USA: Association for Computing Machinery, June 2011, pp. 175–186. ISBN: 978-1-4503-0663-8. DOI: `10.1145/1993498.1993520`.

[Sco13] Michael L. Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2013. ISBN: 978-3-031-00612-8. DOI: `10.1007/978-3-031-01740-7`.

[Sen07] Koushik Sen. "Concolic Testing." In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. New York, NY, USA: Association for Computing Machinery, Nov. 2007, pp. 571–572. ISBN: 978-1-59593-882-4. DOI: `10.1145/1321631.1321746`.

[SFC92] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. "Formal Specification of Memory Models." In: *Scalable Shared Memory Multiprocessors*. Ed. by Michel Dubois and Shreekant Thakkar. Boston, MA: Springer US, 1992, pp. 25–41. ISBN: 978-1-4615-3604-8. URL: https://doi.org/10.1007/978-1-4615-3604-8_2.

[Sho+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2011. ISBN: 978-3-031-01733-9. DOI: 10.1007/978-3-031-01733-9.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C." In: *ACM SIGSOFT Software Engineering Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750.

[SPA92] SPARC International, Inc. "The SPARC Architecture Manual Version 8." In: (1992).

[Ste21] Alan Stern. *Re: Potentially Broken Address Dependency via Test_bit() When Compiling With Clang - Alan Stern*. Oct. 2021. URL: https://lore.kernel.org/all/20211028143446.GA1351384@rowland.harvard.edu/ (visited on 10/18/2023).

[Ste22] Alan Stern. *Re: [PATCH RFC] Tools/Memory-Model: Adjust Ctrl Dependency Definition - Alan Stern*. June 2022. URL: https://lore.kernel.org/all/YqnpshlsAHg7Uf9G@rowland.harvard.edu/ (visited on 10/16/2023).

[syza] The syzkaller Authors. *syzkaller - Found Bugs*. Source Code. URL: https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md (visited on 10/18/2023).

[syzb] The syzkaller Authors. *syzkaller Documentation - Setup: Ubuntu Host, QEMU vm, X86-64 Kernel*. URL: https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md (visited on 11/07/2023).

[syzc] The syzkaller authors. *syzkaller - Kernel Fuzzer*. Source Code. URL: https://github.com/google/syzkaller (visited on 10/27/2023).

[Tor12] Linus Torvalds. *Re: Memory Corruption Due to Word Sharing*. Feb. 2012. URL: https://gcc.gnu.org/legacy-ml/gcc/2012-02/msg00013.html (visited on 10/14/2023).

[Tor14] Linux Torvalds. *Re: [RFC][PATCH 0/5] Arch: Atomic Rework - Linus Torvalds*. Feb. 2014. URL: `https://lore.kernel.org/all/CA+55aFwi-wd_heT_R-ngHAS5=aVNqy4PqKTriK4bVWpuS5+pkg@mail.gmail.com/` (visited on 10/18/2023).

[Tur37] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem." In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. ISSN: 1460-244X. DOI: `10.1112/plms/s2-42.1.230`.

[WG118] WG14. *C Standard - Early C2x Draft*. 9899. Nov. 2018. URL: `https://www.open-std.org/JTC1/SC22/WG14/www/docs/n2310.pdf` (visited on 09/10/2023).