

Should Linux Kernel Memory Model Dependencies Be Afraid of a Big Bad Optimizing Compiler?

Sebastian Reimers

TU Munich
Munich, Germany
sebastian.reimers@tum.de

Martin Fink

TU Munich
Munich, Germany
martin.fink@cit.tum.de

Marco Elver

Google
TU Munich
Munich, Germany
melter@kernel.org

Hernán Ponce de León

Huawei
Munich, Germany
hernanl.leon@huawei.com

Dennis Sprokholt

TU Munich
Munich, Germany
dennis.sprokholt@tum.de

Pramod Bhatotia

TU Munich
Munich, Germany
pramod.bhatotia@tum.de

Abstract

The reliability of the global compute stack rests on an informal symbiosis between the Linux kernel and its compilers. To scale on weak memory architectures, the kernel relies on fine-grained dependency ordering, a zero-overhead synchronization mechanism preserved by data or control flow. However, a “compiler gap” has emerged: while the Linux Kernel Memory Model (LKMM) formalizes these dependencies, modern compilers optimize under C11, which does not recognize them. As optimizations grow more aggressive, this mismatch risks silently eliminating critical synchronization.

We present a novel hybrid framework for automatically detecting broken LKMM dependencies at scale in the Linux kernel. It bridges formal semantics with system-level verification through: (1) a rigorous operational characterization of dependencies; (2) a path-sensitive compiler-based analysis integrated into LLVM to identify broken dependency chains; and (3) a model checking phase to prove actual violations.

Our analysis of the v6.16 kernel, covering 25,000+ modules and 2 million dependency chains, gives us a high degree of confidence that the kernel remains semantically safe, even after nearly half of all syntactic dependencies are broken. This safety is not due to formal guarantees but rather “defense-in-depth”: a combination of inline assembly, locks, and luck! This work provides the empirical foundation to resolve a decade-long debate and paves the way for concurrency-aware compilation.

CCS Concepts: • **Software and its engineering** → **Operating systems**; **Multithreading**; **Model checking**; **Auto-**

mated static analysis; *Concurrent programming languages*; **Semantics**; **Compilers**; • **Theory of computation** → **Axiomatic semantics**.

Keywords: Weak memory consistency models; Linux kernel; Formal verification; Translation validation; Compilers

ACM Reference Format:

Sebastian Reimers, Martin Fink, Marco Elver, Hernán Ponce de León, Dennis Sprokholt, and Pramod Bhatotia. 2027. Should Linux Kernel Memory Model Dependencies Be Afraid of a Big Bad Optimizing Compiler?. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27)*, April 11–15, 2027, Heraklion, Greece. ACM, New York, NY, USA, 23 pages. <https://doi.org/10.1145/3845814.3850075>

1 Introduction

Shared-memory multiprocessors are now ubiquitous, from mobile devices to hyperscale datacenters. Yet, writing efficient parallel code remains challenging, especially in the Linux kernel, where every cycle matters. On *weaker* architectures such as Arm [5, 47] and Power [18], kernel developers rely on fine-grained *dependency ordering*. Unlike memory barriers (fences), which flush store buffers or stall pipelines [45], dependency ordering exploits implicit hardware guarantees. When loading an address from memory and then loading from it, hardware respects the order between these accesses. This “free” synchronization underpins scalable kernel primitives, e.g., **Read-Copy-Update (RCU)** [50, 56], which provides low-overhead reader-writer synchronization.

Kernel on weak memory architectures: Dependency ordering gives the kernel zero-cost synchronization on weak architectures—but only if compilers preserve it.

Crucially, hardware guarantees must be expressible in C in order to profit from them. In 2012, Torvalds explicitly rejected C11’s memory model for the kernel [68], stating:

“[...] the kernel will happily take the memory model of the underlying hardware [...]”

ASPLOS '27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).

This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27)*, April 11–15, 2027, Heraklion, Greece, <https://doi.org/10.1145/3845814.3850075>.

rather than conform to a standard that did not grant developers sufficient control [69]. A deliberate bet that forgoes language support in favor of performance.

The core issue is a fundamental mismatch between the Linux kernel’s dependency-based ordering and guarantees of modern C/C++ [4]. For standard-compliant compilers, unless a dependency is explicitly marked with a specific atomic operation (such as `memory_order::acquire`), it does not exist as an optimization constraint. C11 proposed a lighter alternative, `memory_order::consume`, enforcing ordering through hardware dependencies instead of barriers.

In practice, however, the implementation of *consume* has been a notorious failure. Because C11 compilers do not treat syntactic dependencies as ordering constraints, they often break during optimizations. As Boehm [13] summarized:

“[...] it is widely accepted that the current definition of `memory_order::consume` in the standard is not useful. All current compilers essentially map it to `memory_order::acquire` [...]”

Consequently, the kernel is left without a faithfully implemented language-level model to express the very dependencies upon which its performance and correctness rely.

C11 standard not suitable for Linux: C11’s only mechanism for dependency ordering was never implemented as intended; no language-level support exists.

The problem: Compilers (likely) break dependencies.

From the compiler’s perspective, ignoring dependencies enables more aggressive optimization and reordering. Outside *consume*, ordering does not rely on dependencies in standard C/C++. However, this can introduce correctness bugs in the Linux kernel. As McKenney [52] noted in 2014:

“The thing that makes me very nervous is how much the definition of ‘reasonable optimization’ has changed. [...] Are ARM and Power really the bad boys here? Or are they instead playing the role of the canary in the coal mine?”

An optimization is only “reasonable” when it does not cause the program to show new behavior [73]. Yet what is reasonable for standard C/C++ may not be for the kernel.

By 2019, this became explicit: the kernel began actively disabling compiler optimizations, as noted by Torvalds [70]:

“We actually disable lots of “valid” (read: the standard allows them, but they are completely wrong for the kernel) optimizations because they are wrong.”

A more recent example comes from the arm64 maintainers: in LTO builds, all `READ_ONCE()` accesses were promoted to acquire semantics, sacrificing the performance advantage that dependency ordering was designed to provide.

Kernel vs. compilers: The kernel and compilers are in open conflict: the kernel actively suppresses valid optimizations to survive.

Although dependencies are well-understood for architectures [5, 9], that is not the case for programming languages or compilers. Formal memory models of standard C/C++ become unbelievably complex regarding dependencies (used for *consume*) [11], or do not model them [41, 73].

The **Linux Kernel Memory Model (LKMM)** faces the same issue: `memory-barriers.txt` [35] captures the “*the minimum functionality that one can rely on for any particular barrier*”, which offers an intuition of dependencies, but is crucially *no specification*. Alglave et al. [7] later formally defined the memory model in the cat language, which captures most intuitions of kernel developers but is incomplete regarding dependencies, even leading to a recent revision [60].

Fragmented specifications: The kernel’s memory model is fragmented—part informal prose, part axiomatic formalization—leaving no authoritative specification a compiler could attempt to implement.

The research gap: No concrete evidence. Both kernel and compiler developers recognize the problem exists. In 2020, Deacon convened a dedicated Linux Plumbers Conference (LPC) session on the problem [20]. Based on proposed ideas, Zijlstra proposed a new kernel macro in June 2021: `volatile_if()` [2], designed to force compilers to respect control dependencies. His opening was blunt:

“With optimizing compilers becoming more and more aggressive and C so far refusing to acknowledge the concept of control-dependencies even while we keep growing the amount of reliance on them, things will eventually come apart.”

Torvalds rejected the proposal the same day [71], dismissing it as “*voodoo programming*”, demanding empirical proof:

“[...] but dammit, I want to see an actual real example arguing for why it would be relevant and why the compiler would need our help”

At LPC 2021, Deacon, Zijlstra, McKenney, and Alglave convened again under the telling title “*The Never-Ending Saga of Control Dependencies*” [21]. Again, no resolution was reached. In 2022, Heidekrüger and Elver presented a *status report* at LPC [34], identifying a concrete broken dependency, yet still unable to prove that it caused an actual semantic violation.

In 2024, the C++ standards committee, formally proposed to deprecate `memory_order::consume` [13] after thirteen years in which no compiler had implemented it as intended, leaving the kernel as the only user of dependencies in C/C++.

The problem is not confined to C: as of 2026, Rust-for-Linux maps relaxed atomics directly to `READ_ONCE()` [26], inheriting the same ordering assumptions.

The gap remains: A decade of proposals, never-ending debates, workarounds, and deprecations has produced no solution—the gap between kernel semantics and compiler behavior remains open.

This leads to the central question of this paper:

Research question

To what extent do modern optimizing compilers break LKMM dependency chains, and can a framework finding these breakages scale to real production code?

What the community lacked was not awareness of the problem, nor formal models, nor proposed mitigations—it lacked the concrete empirical proof Torvalds had demanded. Detecting such examples is a challenging task. A dependency is only “broken” in a meaningful sense if its removal severs LKMM’s preserved-program-order, i.e., if no other synchronization (such as fences or locks) preserves the intended ordering. Furthermore, the sheer scale of the kernel—exceeding 30 million lines of code—renders traditional formal verification and global model checking computationally infeasible.

Our approach and contributions. We present a novel hybrid framework to detect broken LKMM dependencies at the scale of the full Linux kernel. Our approach bridges the gap between high-level static analysis and low-level formal verification through a three-pillared pipeline:

- **Formal operational semantics:** We clearly define how dependencies are generated over a key subset of the C programming language, which guides our analyses. This formal foundation allows us to precisely define when an optimization *breaks* a dependency (§ 4).
- **Compiler-based filtering at scale:** We hook into the LLVM pipeline pre- and post-optimizations. Through differential analysis across 25,000+ modules, we detect broken syntactic dependencies and identify a manageable set of “at-risk” fragments (§ 5) from millions of kernel lines.
- **Model-checked verification:** To eliminate false positives, we employ bounded model checking using DARTAG-NAN [29]. For each fragment at risk, we verify it against the formal LKMM memory model. Any execution failing verification is the empirical proof required to resolve the decade-long debate in the community (§ 6).

After analyzing 25,000+ modules and almost 2 million dependency chains, we are confident that the current implementation of the Linux kernel is *safe* from intra-thread reorderings caused by broken dependencies, when compiled to x86-64 or Arm. This safety, however, is built on fragile knowledge of compiler implementations rather than robust guarantees.

Artifact availability. The entire toolchain with the formal semantics will be publicly available.

2 Background

Language Memory Consistency Models. Memory models define which values loads can observe. This depends on when writes become visible and which reorderings hardware or compilers may perform. They are commonly formalized in two ways: (1) operational models, which describe thread

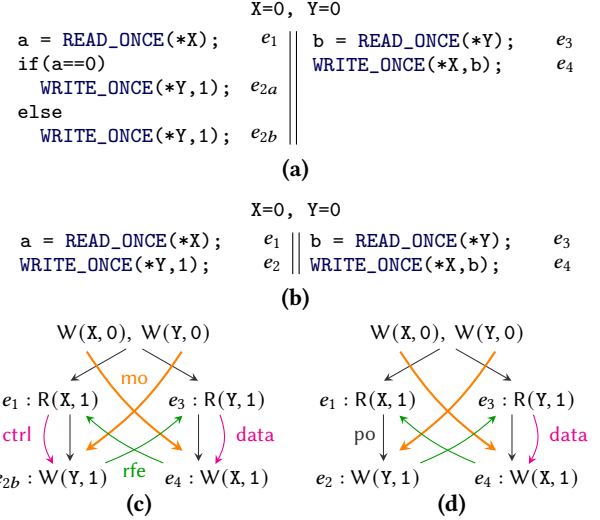


Figure 1. A transformation that breaks dependencies under LKMM. In (c), the `ctrl`-dependency on the left thread orders e_1 before e_2 . In contrast, in (d), event e_1 observes the value written by e_2 , terminating with $a=1$. Note that two writes e_{2a} and e_{2b} in (a) are merged to event e_2 after optimizations. (All accesses are *marked*)

and memory behavior step by step, and (2) axiomatic models, which constrain relations between operations in execution graphs. Both the LKMM and RC11¹ are axiomatic [7, 8, 41]. The orderings enforced by primitives differ between them.

Execution graphs. In axiomatic semantics, program behavior is characterized by a set of execution graphs. Each graph consists of events, or *operations* in Linux kernel terminology. Events are reads R, writes W, or fences F. Some events, namely reads and writes, can be explicitly *marked* as atomic through `READ_ONCE`, `WRITE_ONCE`, or `atomic_*`, while regular memory access (i.e. through pointer dereference) are *plain*. Relations between operations include *program order* `po`, *read-from* `rf`, *modification order* `mo`, and the *read-modify-write* relation `rmw`. There are also dependency-based orderings [35]:

- A *data dependency* `data` exists between a read and a po-later write if the read value determines the written value.
- An *address dependency* `add` exists between a read and a po-later memory operation if the loaded value is used to compute its address.
- A *control dependency* `ctrl` exists between a load and a po-later memory operation if a conditional is derived from the loaded value and the memory operation is in a branch guarded by said conditional. This differs from control dependencies on hardware, where the dependency extends beyond the scope of the conditional [48].

¹In this paper we follow the RC11 memory model [41] when reasoning formally about C/C++.

$$\begin{aligned}
\text{hb} &\triangleq [\mathbb{M}];(\text{ppo} \cup \text{rfe} \cup \dots);[\mathbb{M}] \\
\text{ppo} &\triangleq \text{to-r} \cup \text{to-w} \cup (\text{fence} \cap \text{int}) \\
&\quad \cup (\text{po-unlock-lock-po} \cap \text{int}) \\
\text{to-r} &\triangleq \text{addr};[\mathbb{R}] \cup \text{dep};[\mathbb{M}];\text{rfi} \\
\text{to-w} &\triangleq (\text{addr} \cup \text{data} \cup \text{ctrl});[\mathbb{W}] \\
&\quad \cup ((\text{mo} \cup \text{fr}) \cap \text{int}) \\
&\quad \cup \text{addr};[\mathbb{P}];\text{wmb} \\
\text{fence} &\triangleq \text{wmb} \cup \text{rmb} \cup \dots \\
\text{po-unlock-lock-po} &\triangleq \text{po};[\mathbb{U}];(\text{po} \cup \text{rf});[\mathbb{LKR}];\text{po}
\end{aligned}$$

Figure 2. Part of LKMM axiomatic model [7, 8].

The relations above are *primitive*. They are established from the program syntax, meaning they can be statically computed. Fig. 1 shows an example program and one possible execution graph including primitive relations.

In axiomatic models, further relations are defined using relational algebra. For example, LKMM (see Fig. 2) defines its *preserved-program-order* **ppo**, relating some po-earlier instruction to some po-later instruction, *i.e.* $\text{ppo} \subseteq \text{po}$. It captures which same-thread operations are guaranteed to remain ordered. For example, it orders marked R/W-R pairs through address or data dependencies (to-r), and marked R/W-W pairs through address, data, and control dependencies (to-w). It also includes order enforced by atomics, fences, and locking primitives.

Consistent executions. Axiomatic semantics restrict the set of *consistent* execution graphs with constraints (commonly called *axioms*) over relations. LKMM defines a notion of *happens-before* **hb**, and the axiom of interest states that consistent executions cannot have an **hb**-cycle.

Consider the C program in Fig. 1a and its execution graph in Fig. 1c containing the cycle: $e_1 \xrightarrow{\text{ctrl}} e_2 \xrightarrow{\text{rfe}} e_3 \xrightarrow{\text{data}} e_4 \xrightarrow{\text{rfe}} e_1$. Using the definitions in Fig. 2, those edges produce the cycle $e_1 \xrightarrow{\text{hb}} e_2 \xrightarrow{\text{hb}} e_3 \xrightarrow{\text{hb}} e_4 \xrightarrow{\text{hb}} e_1$. Thus, this behavior is not allowed by LKMM. While LKMM intends to preserve the **ppo** orderings, compilers can break (syntactic) dependencies. To illustrate this, consider the program transformation shown between Fig. 1a and 1b. When the program contains the *if-else* construct and the graph the **ctrl** edge, LKMM forbids the execution where e_1 (indirectly) observes the value written by e_2 . However, the transformation breaks the **ctrl**-dependency in the execution graph, which in turn breaks the **hb**-cycle. Once the cycle is gone, the execution is allowed by LKMM. This shows the transformation is unsound according to LKMM.

3 Overview

The Linux kernel uses address, data, and control dependencies for ordering on weak architectures without costly barriers. However, modern compilers optimize under RC11, which ignores these dependencies, creating a *compiler gap*: optimizations such as **Link Time Optimization (LTO)** can silently break mechanisms like **RCU**. Fig. 1 shows a transformation

correct under RC11 but invalid under LKMM. Yet, there is no evidence this causes observable misbehavior. We *automatically* validate whether Clang breaks LKMM dependency-based ordering *at scale*: (i) without annotations, and (ii) on full kernel builds rather than litmus tests. Specifically, we look for dependencies that provide **ppo** ordering in LKMM. Since ordering can also come from memory barriers or locks, we also tackle the problem of identifying cases in which dependencies are the *only* ordering guarantee, in addition to finding them in the first place.

Core challenge: Unpreserved program order. LKMM provides a formal abstraction to reason about a program's behavior, but is neither authoritative nor complete. Detecting **ppo** violations is hard for two reasons: First, dependencies are subtle and inconsistently defined across languages, LKMM, and hardware [35, 48]. Second, scalable analysis is difficult: checking **ppo** violations requires reasoning over all executions, which does not scale to the Linux kernel.

Core idea #1: Formalize dependencies. We analyze existing examples and informal definitions of dependencies [2, 22, 35] and define an operational semantics of how dependencies are generated under LKMM. The semantics are defined directly on structured C control flow syntax, allowing us to capture the subtle generation of **ctrl**-dependencies. Our analysis builds on those formal foundations.

Core idea #2: Hybrid analysis. To address scalability, we combine compiler-based analysis and bounded model checking. The compiler-based analysis validates that many dependencies are never broken. Remaining cases are reduced to small program fragments and analyzed via model checking, drastically limiting the number of executions to consider.

Semantics. LKMM defines dependencies axiomatically, describing allowed execution graphs rather than how dependencies arise. It exists in both informal prose [35] and formal models [7, 8], and diverges from hardware in key ways (e.g., dependencies through memory, differing control-dependency scopes), making it difficult to implement an analysis.

Thus, we define an operational semantics that captures how loads, stores, and control flow generate dependencies at runtime (Sec. 4), forming the basis of our hybrid design.

Compiler-based analysis: filtering at scale. A whole-kernel compiler-based analysis extracts dependencies before and after optimization and flags them as *definitely preserved* or *potentially broken*. This reduces the search space from the entire Linux kernel to small code fragments with *potentially* broken dependencies.

Model checking: precise verification. Model checking then determines whether these candidates correspond to real violations or false positives, accounting for alternative orderings through locks and barriers. While model checking alone usually does not scale beyond small tests, restricting it to fragments enables precise analysis on real kernel code.

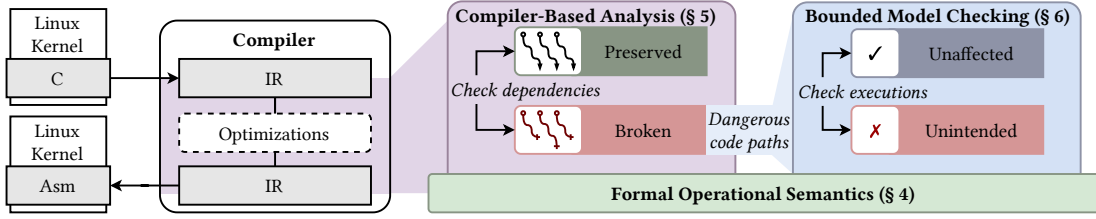


Figure 3. System overview. A compiler-based analysis identifies broken dependencies at compile time. Bounded model checking verifies whether any executions rely on them. Both analyses are built on our formal operational semantics of dependencies.

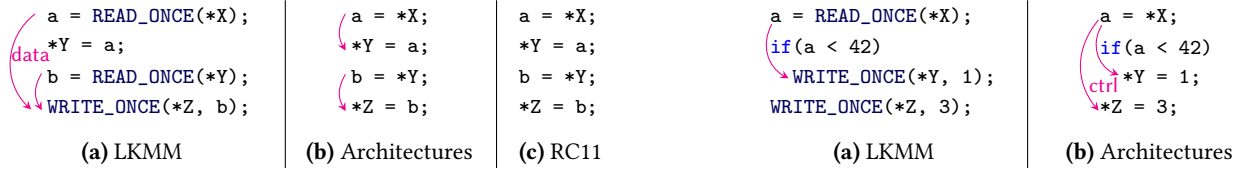


Figure 4. LKMM *data* dependencies differ from RC11 [42], Arm, POWER, and RISC-V [48].

4 LKMM Dependencies

Before implementing any algorithm that detects dependencies, we must precisely define what they are. The kernel community relies on dependencies as described in the “*collective opinion of its maintainers*” [35], upon which hardly any algorithm can be built. Instead, we propose a mathematically rigorous characterization of dependencies in LKMM. To be clear, we do not insist on our particular formalization, but we do insist it conforms to *some* semantics.

4.1 Intuition

We begin by contrasting dependencies in different models. For the examples below, architectures disagree with LKMM.

Marked vs. unmarked. LKMM generates dependencies between any accesses, but only orders between *marked* ones. In contrast, RC11 has none, while architectures have them between any accesses. Fig. 4 illustrates this difference.

Structured control flow. LKMM models dependencies on C syntax with structured control flow such as *if-else* statements, while architectures use branch instructions. Fig. 6 shows that LKMM does not generate *ctrl* dependencies after the *if* body [35, 53], while architectures do.

Through memory. Unlike architectures [48], LKMM dependencies *pass through memory*, shown in Fig. 4a. C does not specify whether thread-local variables reside in registers or on the stack; hence, both should generate dependencies identically [60]. Thus, LKMM expects *data*-dependencies through memory, something architectures do not follow.

Very syntactic dependencies. *Syntactic* dependencies derive purely from program syntax [40, 63]. Historically, architectures used semantic dependencies, which could be *false* (e.g., after multiplication by zero), and proved difficult

to formalize. Hence, syntactic dependencies were adopted instead, consequently used by LKMM as well.

As LKMM dependencies go *through memory*, new scenarios for false dependencies arise. In Fig. 5, another thread affects if a value truly depends on a read. Our interpretation *explicitly includes* this class of false dependencies, in line with the historical choice to include false dependencies in syntactic dependencies.

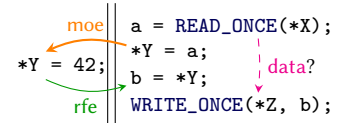


Figure 5. As LKMM dependencies pass through memory, arbitrary threads could cause the final write to not “truly depend” on the initial read.

4.2 Semantics

As dependencies differ subtly between languages, we explicitly formalize dependencies in LKMM, upon which we build our analysis algorithms (in Secs. 5 and 6).

Syntax. As the Linux kernel is implemented in C, we consider its syntax. We only use a core subset with interesting properties for dependencies (Fig. 7). We omit *do-while* statements; simplify arithmetic to only integer addition, omitting fixed-width integers and floating-point numbers; and disallow assignments as expressions. This does not affect characterization of dependencies, but simplify their presentation. We explicitly represent adjacent statements with *concat* and capture loads/stores with *load_u/load_m* for unmarked/ marked loads and *store_ustore_m* for unmarked stores.

Operational semantics. We define an operational semantics describing the generated dependencies over the program syntax in Fig. 8. As dependencies may rely on concrete values, e.g., when dereferencing computed addresses, these must be

<code>stmt ::= while expr stmt</code>	<code>expr ::= const $\mathbb{Z}$</code>
<code> if expr stmt stmt</code>	<code> var name</code>
<code> assign name expr</code>	<code> add expr expr</code>
<code> concat stmt stmt</code>	<code> load_m expr</code>
<code> store_m expr expr</code>	<code> load_u expr</code>
<code> store_u expr expr</code>	<code>name ::= ...</code>

Figure 7. Core of C syntax needed for dependencies.

captured explicitly. We capture dependencies by relating a marked load (`load_m`) and marked store (`store_m`) by their unique runtime⁴ identifier, denoted with their `@id` suffix.

Environments. Γ is the context with variables and M is memory indexed by addresses—e.g., $\Gamma = \{a \mapsto 3\}$, $M = \{4 \mapsto 42\}$. Those concrete values are needed because they affect dependencies, e.g., to compute a memory address or branch condition. Γ_d , M_d , and S_c track dependencies. Γ_d captures for a variable which `load_m` expressions (referenced by id) have produced its values—e.g., $\Gamma_d = \{a \mapsto \{\text{id}\#3, \dots\}\}$. M_d captures `load_m` expressions for memory—e.g., $M_d = \{20 \mapsto \{\text{id}\#7\}, \dots\}$. S_c tracks sources for `ctrl` dependencies, consisting of the `load_m` expressions that affected the nested if-conditions—e.g., $S_c = \{\text{id}\#2, \text{id}\#8\}$.

Evaluation relations. The rules cover several evaluation relations, most notably $\llbracket \cdot \rrbracket_{\text{dep}}^{\text{expr}}$ and $\llbracket \cdot \rrbracket_{\text{dep}}^{\text{stmt}}$, which track dependencies through expressions (`expr`) and statements (`stmt`), respectively. To explain how we track dependencies, we explain the inputs and outputs of each evaluation relation.

- $\llbracket \cdot \rrbracket_{\text{dep}}^{\text{stmt}}$ computes the dependencies for a statement. It takes (1) the environments $\langle \Gamma, M, \Gamma_d, M_d, S_c \rangle$ and (2) the executed statement as inputs. As output it produces the (1) updated environments $\langle \Gamma', M', \Gamma'_d, M'_d \rangle$, which *excludes* S_c as they never change after statements; and (2) the generated `data`-, `addr`-, and `ctrl` dependencies.
- $\llbracket \cdot \rrbracket_{\text{dep}}^{\text{expr}}$ computes the dependencies for an expression. It takes the same inputs as $\llbracket \cdot \rrbracket_{\text{dep}}^{\text{stmt}}$, with an expression instead of a statement. It returns (i) the *data sources* (often denoted D), which are a set of marked loads (referenced by id) included in or referenced by the expression; (ii) `addr`-dependencies to the marked loads in the expression; and (iii) `ctrl`-dependencies to those same marked loads.
- $\llbracket \cdot \rrbracket_{\text{eval}}$ evaluates an expression to a value, which is needed to determine computed addresses or branch conditions.

Rules. Fig. 8 contains several interesting rules, covering marked accesses, their interaction with the local environment, and basic control flow. The remaining definitions for the full syntax in Fig. 7 are included in App. C.

- **ASSIGN** generates the `addr`- and `ctrl`-dependencies for an expression. It also obtains the *data sources* D , which it includes in the local environment Γ_{deps} . Notably, it only

generates `addr`- and `ctrl`-dependencies to the marked loads – `data`-dependencies *never* target loads.

- **IF-TRUE/FALSE** evaluates the condition and generates `data`-, `addr`-, and `ctrl`-dependencies from the taken branch. Notably, to obtain `ctrl` dependencies, it propagates the data sources D_c of its condition to the bodies.
- **STORE-M** generates all `data`-, `addr`-, and `ctrl`-dependencies to a marked store. In addition, it stores its data source to memory, which subsequent loads may propagate. It obtains data sources D_a and D_v from its address (e_a) and value (e_v) expressions respectively. Those generate `addr` and `data` dependencies to the marked store.
- **LOAD-M** generates `addr`- and `ctrl`-dependencies to a marked load. While loads never generate `data` dependencies, they do act as data sources. It additionally propagates the data sources from memory for its location, includes itself as a data source and the sources of its address expression.

5 Compiler-Based Analysis

Optimizing compilers may eliminate syntactic dependencies. If these are required for correct ordering under **LKMM**, their removal may cause bugs. Using Sec. 4, we present an analysis to detect broken dependencies. Fig. 1 shows a program pre- and post-optimizations. All dependencies should remain, yet the `ctrl` dependency in Fig. 1c is absent in Fig. 1d, i.e., broken.

5.1 Assumptions

For soundness, we use the assumptions from Sec. 4.1: syntactic dependencies *always* exist (including false dependencies).

Thread-local analysis. Dependencies are thread-local, thus we can analyze threads individually. As discussed, this assumes that no other thread breaks a `data`; `rfi` propagation. Especially during compilation, the analysis only sees a single module at a time. Externally defined code, linked later, is opaque but could lead to false dependencies.

Absence of front- and backend-optimizations. Optimizing compilers represent a program through different **Intermediate Representations** (IRs). For our analysis to run pre- and post-optimizations, it must be implemented on an **IR** as well. Consequently, the analysis cannot cover two conversions: (i) A frontend conversion from the C sources to the compiler **IR**, and (ii) backend conversions from the **IR** to the final target assembly. We must thus assume that both conversions are straightforward and do not apply heavy optimizations, which for our implementation (App. B) is the case.

5.2 Finding Dependencies

The analysis’s goal is to determine syntactic dependencies between a marked load and another marked access.

Propagating operations. Any marked load acts as a data source. Operations using such a value propagate the dependency. For *syntactic* dependencies, analysis is reduced to tracking data sources that can reach another marked access.

⁴We assume identifiers to be globally unique per execution. Repeated executions of “the same” instruction within a loop have different identifiers.

$$\begin{array}{c}
\frac{\langle \langle \Gamma, M \rangle, e \rangle \Downarrow_{\text{eval}} v \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D, a, c \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{assign } n e \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma[n \mapsto v], M, \Gamma_d[n \mapsto D], M_d \rangle, \langle \emptyset, a, c \rangle \rangle} \text{ASSIGN} \\
\\
\frac{\langle \langle \Gamma, M \rangle, e_c \rangle \Downarrow_{\text{eval}} v_c \quad v_c \neq 0 \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_c \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_c, a_c, c_c \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \cup D_c \rangle, s_{\top} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_s, a_s, c_s \rangle \rangle} \text{IF-TRUE} \\
\frac{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{if } e_c s_{\top} s_{\perp} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_s, a_c \cup a_s, c_c \cup c_s \rangle \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, s_0 \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_1, a_1, c_1 \rangle \rangle} \\
\frac{\langle \langle \Gamma', M', \Gamma'_d, M'_d, S_c \rangle, s_1 \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma'', M'', \Gamma''_d, M''_d \rangle, \langle d_2, a_2, c_2 \rangle \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{concat } s_0 s_1 \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma'', M'', \Gamma''_d, M''_d \rangle, \langle d_1 \cup d_2, a_1 \cup a_2, c_1 \cup c_2 \rangle \rangle} \text{CONCAT} \\
\\
\frac{\langle \langle \Gamma, M \rangle, e_a \rangle \Downarrow_{\text{eval}} a \quad \langle \langle \Gamma, M \rangle, e_v \rangle \Downarrow_{\text{eval}} v \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_a \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_a, a_a, c_a \rangle \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_v \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_v, a_v, c_v \rangle}{d_m = \{ \langle d_v, \text{id} \rangle \mid d_v \in D_v \} \quad a_m = \{ \langle d_a, \text{id} \rangle \mid d_a \in D_a \} \quad c_m = \{ \langle s_c, \text{id} \rangle \mid s_c \in S_c \}} \text{STORE-M} \\
\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, (\text{store_m } e_a e_v) @ \text{id} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma, M[a \mapsto v], \Gamma_d, M_d[a \mapsto D_a \cup D_v] \rangle, \langle d_m, a_v \cup a_a \cup a_m, c_v \cup c_a \cup c_m \rangle \rangle \\
\\
\frac{}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{var } n \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle \Gamma_d[n], \emptyset, \emptyset \rangle} \text{VAR} \\
\\
\frac{\langle \langle \Gamma, M \rangle, e_a \rangle \Downarrow_{\text{eval}} a \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_a \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_a, a_a, c_a \rangle \quad a_m = \{ \langle d_a, \text{id} \rangle \mid d_a \in D_a \} \quad c_m = \{ \langle s_c, \text{id} \rangle \mid s_c \in S_c \}}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, (\text{load_m } e_a) @ \text{id} \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle M_d[v] \cup D_a \cup \{\text{id}\}, a_v \cup a_m, c_v \cup c_m \rangle} \text{LOAD-M}
\end{array}$$

Figure 8. Partial operational semantics capturing how **addr**-, **data**-, and **ctrl**-dependencies are generated (extended in App. C).

$$\begin{array}{c}
\frac{\langle \langle P, \Gamma_d, M_d, S_c \rangle, e_a \rangle \Downarrow_{\text{sa}}^{\text{expr}} \langle D_a, a_a, c_a \rangle \quad a_m = \{ \{ \langle \text{PUP}_s, d_a, \text{id} \rangle \mid \langle P_s, d_a \rangle \in D_a \} \} \quad c_m = \{ \{ \langle P, s_c, \text{id} \rangle \mid s_c \in S_c \} \}}{\langle \langle P, \Gamma_d, M_d, S_c \rangle, (\text{load_m } e_a) @ \text{id} \rangle \Downarrow_{\text{sa}}^{\text{expr}} \langle M_d[e_v] \cup D_a \cup \{ \langle P, \text{id} \rangle \}, a_a \cup a_m, c_a \cup c_m \rangle} \text{LOAD-M} \\
\\
\frac{\langle \langle P, \Gamma_d, M_d, S_c \rangle, e_v \rangle \Downarrow_{\text{sa}}^{\text{expr}} \langle D_v, a_v, c_v \rangle \quad \langle \langle P, \Gamma_d, M_d, S_c \rangle, e_a \rangle \Downarrow_{\text{sa}}^{\text{expr}} \langle D_a, a_a, c_a \rangle \quad d_m = \{ \{ \langle \text{PUP}_s, d_v, \text{id} \rangle \mid \langle P_s, d_v \rangle \in D_v \} \} \quad a_m = \{ \{ \langle \text{PUP}_s, d_a, \text{id} \rangle \mid \langle P_s, d_a \rangle \in D_a \} \} \quad c_m = \{ \{ \langle P, s_c, \text{id} \rangle \mid s_c \in S_c \} \} \quad D = \{ \langle \text{PUP}_s, d \rangle \mid \langle P_s, d \rangle \in D_v \cup D_a \}}{\langle \langle P, \Gamma_d, M_d, S_c \rangle, (\text{store_m } e_a e_v) @ \text{id} \rangle \Downarrow_{\text{sa}}^{\text{stmt}} \langle \langle \Gamma_d, M_d[e_a \mapsto D] \rangle, \langle d_m, a_v \cup a_a \cup a_m, c_v \cup c_a \cup c_m \rangle \rangle} \text{STORE-M} \\
\\
\frac{\langle \langle P, \Gamma_d, M_d, S_c \rangle, e_c \rangle \Downarrow_{\text{sa}}^{\text{expr}} \langle D_c, a_c, c_c \rangle \quad \langle \langle P \cdot (\langle \text{id}, \top \rangle), \Gamma_d, M_d, S_c \cup D_c \rangle, s_{\top} \rangle \Downarrow_{\text{sa}}^{\text{stmt}} \langle \langle \Gamma'_d, M'_d \rangle, \langle d_{s_{\top}}, a_{s_{\top}}, c_{s_{\top}} \rangle \rangle \quad \langle \langle P \cdot (\langle \text{id}, \perp \rangle), \Gamma_d, M_d, S_c \cup D_c \rangle, s_{\perp} \rangle \Downarrow_{\text{sa}}^{\text{stmt}} \langle \langle \Gamma''_d, M''_d \rangle, \langle d_{s_{\perp}}, a_{s_{\perp}}, c_{s_{\perp}} \rangle \rangle}{\langle \langle P, \Gamma_d, M_d, S_c \rangle, (\text{if } e_c s_{\top} s_{\perp}) @ \text{id} \rangle \Downarrow_{\text{sa}}^{\text{stmt}} \langle \langle \Gamma'_d \cup \Gamma''_d, M'_d \cup M''_d \rangle, \langle d_{s_{\top}} \cup d_{s_{\perp}}, a_c \cup a_{s_{\top}} \cup a_{s_{\perp}}, c_c \cup c_{s_{\top}} \cup c_{s_{\perp}} \rangle \rangle} \text{IF}
\end{array}$$

Figure 9. Compiler-based analysis semantics: key rules for path-indexed dependency construction. Where $(A \cup B)(k) = A(k) \cup B(k)$, with undefined elements in $A(k)$ and $B(k)$ treated as $\emptyset$.

Path-sensitive dependencies. To not create *false* dependencies, we must consider control flow. In Fig. 11a, the **data** dependency between @1 and @3 exists only when c is *false*. To guarantee that found dependencies always exist, we have two options: (i) we do not consider this **data** dependency, or

(ii) we adjust our definition of dependencies.

Since the first option would result in essentially zero dependencies, we define dependencies to be path-sensitive. Concretely, we attach a *path prefix* P to each dependency, recording which branches have been taken to find this depen-

dependency. With this addition, we guarantee that a dependency exists in *all executions that follow its path prefix*.

Underapproximation. Tracking syntactic dependencies by recording operations that use at least one value that is derived from a data source, will always result in a (at least well defined) subset of all possible dependencies. Beyond the thread-local restriction above, compile-time analysis without a global program view introduces three further limitations:

- *External functions* — Functions outside the current compilation unit are opaque, so we conservatively assume dependencies do not carry through them.
- *Pointer aliasing* — Through-memory dependencies require two accesses to use the same address. Static analysis cannot generally prove that two different expressions alias; instead, we require expressions to be identical, which must evaluate to the same address.
- *Loops* — Loops create an unbounded number of paths. For simplicity, we analyze at most one loop iteration, yielding only the paths of zero or exactly one iteration.

5.3 Rules

We define operational semantics over the core C subset (Fig. 7). The rules define dependencies the analysis finds—an underapproximation of Sec. 4 due to the restrictions above.

Compile-time constraints. At compile time, no runtime values exist. Conditions cannot be evaluated, addresses cannot be resolved, variables can have multiple possible values. Instead, we must determine when two expressions evaluate to the same value *in all executions*. Consider the code in Fig. 10. Clearly, the expressions a in the stores to $*Y$ and $*Z$ cannot be treated the same even though they are verbatim identical. After the addition, a also depends on b at the time of the addition (Even if $b == 0$). To capture this, we introduce a map Γ_d from *expressions* to the set of data sources (loads) they depend on.

An assignment $v = e$ updates Γ_d , by changing the data sources of v to the union of data sources of variables used in e (this can also remove data sources). Additionally, *all* expressions in Γ_d using v must be updated, as no future expression can depend on its old value. In general, we consider two expressions to be the same if (i) they are verbatim identical and (ii) they have the same data sources (under the same path).

Path-prefixed environments. To capture path-sensitivity, we introduce the *path prefix* P , recording traversed branches. In our operational semantics, we define a path as

$$P = [(s_1, v_1), \dots, (s_n, v_n)] \mid s \in \text{Stmts}; v \in \{\top, \perp\}$$

Stmts are all control flow statements in a program.

Each environment tracking data sources in the compiler-based analysis must be path-prefixed:

- Γ_d maps expressions to path-prefixed data sources: $\{e \mapsto \{P \mapsto \{\text{id}\#1\}\}\}$ records that a load with $\text{id}\#1$ is a data source for an expression e *only* in executions following

path P . Notably, *all* executions following P .

- Likewise, M_d maps expressions—representing memory locations—to path-indexed data sources.
- S_c is a map of paths to control dependency sources.

Evaluation relations. Our analysis uses two evaluation relations to construct dependencies. The relation $\Downarrow_{\text{dep}}^{\text{stmt}}$ analyzes statements, taking the environments $\langle P, \Gamma_d, M_d, S_c \rangle$ and a statement as inputs. It produces updated environments $\langle P', \Gamma'_d, M'_d \rangle$ (excluding S_c which does not change within statements) and path-prefixed dependencies $\langle d, a, c \rangle$. The relation $\Downarrow_{\text{dep}}^{\text{expr}}$ analyzes expressions, taking the same inputs but producing data sources D and path-indexed dependencies $\langle a, c \rangle$.

<pre> a = READ_ONCE(*X1); // @1 b = READ_ONCE(*X2); // @2 WRITE_ONCE(*Y, a); // @3 a = a + b; WRITE_ONCE(*Z, a); // @4 </pre>	$\Gamma_d = \{a \mapsto \{1\}\}$ $\Gamma_d = \{a \mapsto \{1\}, b \mapsto \{2\}\}$ $\Gamma_d = \{a \mapsto \{1, 2\}, b, \dots\}$
---	--

Figure 10. Overwrite example

Key rules. Fig. 9 presents key rules of the analysis. Focusing on *data* dependencies, the most important rules are: (i) LOAD-M recording marked loads as data sources, and (ii) STORE-M constructing dependencies to a marked store.

Take a simple *data* dependency from @1 to @3 in Fig. 10. STORE-M generates *data* dependencies d_m between the store (@id) and the data sources of the value expression e_v . LOAD-M provides those sources. Note that without control flow, the path P remains empty throughout the analysis.

<pre> a = READ_ONCE(*X); // @1 if(c) r1 = a+1 WRITE_ONCE(*Y, a); // @2 else r1 = a+2 WRITE_ONCE(*Y, a); // @3 WRITE_ONCE(*Y, r1); // @4 </pre>	<pre> a = READ_ONCE(*X); // @1 WRITE_ONCE(*Y, a); // @2, 3 if(c) r1 = a+1 else r1 = a+2 WRITE_ONCE(*Y, r1); // @4 </pre>
(a) Conditional dependencies	(b) A legal optimization

Figure 11. *data* dependencies in diverging execution paths

Let us now add control flow: The program in Fig. 11a has three *data* dependencies from the marked load (@1) to the marked stores (@2, @3, @4). Crucially, no execution contains both dependencies to @2 and @3. The IF rule explores both branches $s_\top$ and $s_\perp$ individually. The marked store @2 is then evaluated in the context of $P_\top = [(\text{if}(c), \top)]$, resulting in a dependency $P_\top \mapsto \{(\text{@1}, \text{@2})\}$. @3 follows similarly.

The IF rule returns the original (empty) path, making later statements unconditional, but merges the data sources $\Gamma'_d \cup \Gamma''_d$. Γ_d for @4 contains *two* entries for $r1$, thus generates *two* *data* dependencies: $\{P_\top \mapsto \{(\text{@1}, \text{@4})\}, P_\perp \mapsto \{(\text{@1}, \text{@4})\}\}$.

Algorithmic implementation. The rules introduced above provide a definition of the dependencies the compiler-based analysis will find, but not how it does so. We give pseudo-algorithms of the analysis in App. C.

5.4 Broken Dependencies

The formalized semantics produce ordering dependencies for a given program. To detect whether optimizations break these dependencies, we must compare the dependencies found in the unoptimized and optimized program.

Differential analysis. We apply our compiler-based analysis to the unoptimized and optimized program, producing two sets of path-prefixed dependencies. Let $\langle d_{pre}, a_{pre}, c_{pre} \rangle$ and $\langle d_{post}, a_{post}, c_{post} \rangle$ denote dependencies from the unoptimized and optimized program respectively. A dependency present in the pre-optimization set but absent from the post-optimization set indicates a potentially broken dependency.

Dependency equivalence. Determining equivalence of two dependencies is non-trivial, as optimizations like hoisting independent accesses out of a conditional (e.g., Fig. 11b) can transform execution paths without breaking dependencies that go through them. We note that the marked store @2, 3 is equivalent to stores @2, @3 before optimizations. All expressions used as operands are identical under any control flow paths taken. In general, for dependencies between identical access pairs across multiple paths, if all path prefixes share a common subset of traversed statements, that subset is irrelevant to the dependency's existence.

Broken dependencies. Using the above notion of equivalence, we establish when a dependency is considered broken. It is important to note that *false positives* exist. Due to the underapproximation in Sec. 5.2, the analysis may miss dependencies in the optimized program, causing dependencies in the unoptimized program to be flagged as broken. *Bounded model checking* filters out these false positives.

5.5 Preserved Ordering without Dependencies

To summarize, this analysis detects broken dependencies that may influence the ordering of accesses under LKMM.

Broken dependencies matter only if no other relation preserves the intended order. Fig. 12a shows two control dependencies that are definitely broken after optimizations (Fig. 12b). However, no execution of the optimized program can reorder the accesses due to the memory barrier `smp_mb()`.

5.6 Limitations

In a real compiler modules are handled independently; the compiler-based analysis can only track dependencies within a single compilation unit. Cross-module optimizations (like LTO), dependencies that span compilation-unit boundaries via function pointers, or callbacks cannot be captured

6 Bounded Model Checking

The compiler-based analysis from Sec. 5 is inherently imprecise: it approximates loops and ignores atomics, fences, and locks, reasoning only about to-r and to-w (under-approximating ppo). The compiler may break dependencies if other relations (e.g., fence or po-unlock-lock-po) maintain order.

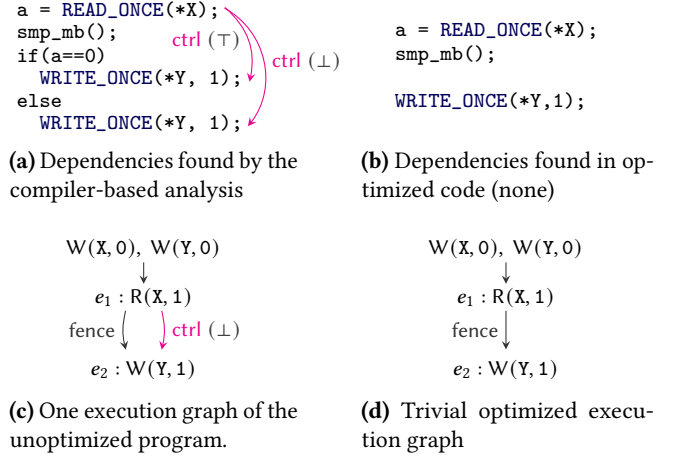


Figure 12. An optimization breaking `ctrl`-dependencies. The order between events is still preserved by a fence.

Existing tools can capture LKMM ordering guarantees and compute ppo precisely [7, 29, 61].

We use bounded model checking to determine whether a reported broken dependency is a false positive (order preserved) or a compiler bug, leveraging the cat language [9].

6.1 Broken Dependencies Axiomatically

To confirm a broken dependency reported by the compiler-based analysis, we must find an execution where the chain's first and last events are not ordered by ppo.

Marking dependent events. Similar to LKMM's data race detection, we add two sets BEGIN and END to cat and populate them using the reported chains. Two events appearing at the start and end of any chain belong to BEGIN and END respectively. By construction, in any execution executing both events, they are related by po.

Characterizing false positives. We say that a chain is a *false positive* if in every execution where both the start and end events are executed, they are ordered by ppo. In order to check for false positives, we define the following relations

$$\begin{aligned} \text{chain} &\triangleq [\text{BEGIN}];\text{po};[\text{END}] \\ \text{chain-dep} &\triangleq [\text{BEGIN}];\text{ppo};[\text{END}] \end{aligned}$$

Guiding analysis along dependency chains. When checking for reorderings, model checking can safely ignore executions where one access is unreachable. We enforce this through an axiom stating that relation $\text{chain} \neq \emptyset$.

Finding truly broken dependencies. Two events $e_1 \in \text{BEGIN}$ and $e_2 \in \text{END}$ can be reordered if they are *not* related by ppo. Therefore, we instruct the model checker to flag any execution where relation chain-dep is empty.

For a dependency reported as broken by the compiler-based analysis, we consider it a false positive if no execution gets flagged through the above axiom.

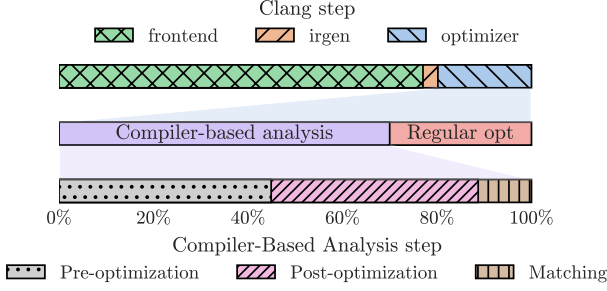


Figure 13. Distribution of the time for compiler-based analysis as a fraction of the total compile time.

6.2 Examples

A real broken dependency. Consider the program in Fig. 1b where the compiler removed the `if-else` and the execution Fig. 1d. Compiler-based analysis reports a broken chain from e_1 to e_2 , so $e_1 \in \text{BEGIN}$ and $e_2 \in \text{END}$. Since the events belong to po, we have $(e_1, e_2) \in \text{chain}$ and thus the relation is non-empty. The events are not related by `ctrl` and, following Fig. 2, we can further derive they do not belong to `ppo` and thus $(e_1, e_2) \notin \text{chain-dep}$. Since no other events are labeled as BEGIN and END, relation chain-dep is empty, and the model checker confirms the dependency is broken.

A false positive. Consider the same program, but containing a barrier before the conditional. Now the optimization would indeed be sound. Even without the `ctrl` edge (Fig. 12d), the events are ordered by the barrier, i.e. $(e_1, e_2) \in \text{fence} \subseteq \text{ppo}$. Now $\text{chain-dep} \neq \emptyset$, and the model checker flags the report by the compiler-based analysis as a false positive.

6.3 Limitations

Opaque code. The entire analysis treats intrinsics (`memcpy`, `memset`, etc.) as opaque. To still track dependencies through them, we exchange calls to those intrinsics with calls to equivalent (but defined) functions, e.g., we exchange `memset` calls with multiple writes of a constant value. Note that expanding intrinsics this way risks traces too large for bounded model checking, as those intrinsics usually span large amounts of memory. Additionally, making their implementation visible in translation units, and potentially inlining them, may trigger further optimizations that cannot occur in practice, introducing false-positives.

Pointer arguments. Inspecting dependency chains from [BEGIN] to [END] in isolation is efficient, but leaves some memory regions uninitialized. During model checking, function arguments may carry pointers to such regions, thus disabling code paths that rely on certain values in memory. Recreating valid memory snapshots in general is an open problem out of scope for this work.

Dependency	Pattern
<code>addr/data</code>	Comparison with non-NULL pointers
<code>ctrl</code>	Identical stores in both branches
<code>ctrl</code>	Compile-time evaluation of the condition
<code>ctrl</code>	Boolean short-circuiting

Table 1. Known code patterns that lead to broken dependencies. Our analyses detect all of these.

7 Implementation and Evaluation

We evaluate our system around three dimensions: (i) a characterization of dependencies in the Linux Kernel, (ii) efficiency, and (iii) accuracy of our hybrid analysis.

Testbed. We compile the Linux kernel to x86-64 and Arm and run our evaluation on a 64-core AMD EPYC 7713P and a 192-core Ampere 1a processor. For brevity, we present the results for x86-64 in the main paper and defer the Arm64 results to App. D.

Kernel configuration. We evaluate the full Linux kernel, version 6.16, with all modules enabled. Other options remain at their defaults except that kernel sanitizers, `lockdep`, and other lock-debugging features that introduce `blockaddress` operations, which DARTAGNAN cannot handle, are disabled.

7.1 Implementation

We first briefly describe the implementation of our hybrid framework; the complete details are covered in App. C. Built on LLVM, the compiler-based analysis runs once on unoptimized and once on optimized IR, using annotations similar to KCSAN [24, 25] to identify kernel primitives that act as marked accesses such as `READ_ONCE` and `rcu_dereference`. To compensate for the loss of high-level structure in LLVM IR, we redefine `ctrl` dependencies using post-dominance, matching the kernel’s intended semantics [51, 54].

We use the DARTAGNAN model checker [19, 29, 32, 33, 61], the only tool handling the full LKMM [9] directly on LLVM IR. To avoid state-space explosion, we isolate minimal kernel fragments per dependency chain, and assume non-deterministic inputs for function parameters. A key challenge is that LLVM IR lacks structured control flow, causing DARTAGNAN to fall back to the hardware notion of `ctrl` instead of the post-dominance-based definition used by LKMM. This can lead to false negatives which we account for in C.

Validation. We validated our implementation against several code patterns that are known to lead to broken dependencies [53, 55]. Our hybrid analysis covers all of these patterns: the compiler-based phase flags the dependency as broken, and the model checker confirms it.

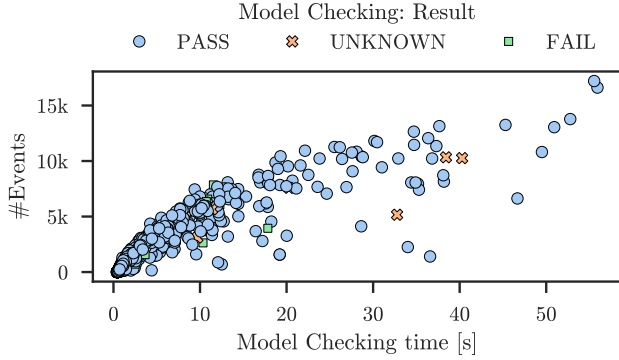


Figure 14. Execution time of DARTAGNAN per chain, relative to the number of events in the analyzed function.

7.2 Evaluation

RQ1. How often are dependencies used in the Linux kernel?

In total, `allmodconfig` results in over 25,000 modules being compiled and analyzed. Approximately 3,000 modules contain at least one dependency chain before optimizations.

The most common sources of dependency chains are headers defining widely used data structures, such as lists and hash tables (`list.h`, `hashtable.h`) and architecture-specific atomic operations (`atomic.h`). Macros like `READ_ONCE` are assigned to their expansion site rather than their definition, so headers like `rwonce.h` do not appear on the list. We provide more details in App. D.

RQ2. Is whole-kernel analysis feasible with the hybrid design?

Compiler-based analysis. Fig. 13 shows the time of each step in our compiler-based analysis. Compiling the full Linux kernel takes 230 minutes, of which 20.6% (48 minutes) is spent in middle-end optimizations. The analysis dominates the optimization runtime with 43 minutes. Pre- and post-optimization passes each account for 44% of the analysis time, with the remaining 12% spent on dependency matching.

Model checking. Fig. 14 shows DARTAGNAN’s execution time (using a 1 min timeout) growing roughly linearly with the number of events in the analyzed function. The median completed run contains 140 events, whereas timed-out runs (not shown) have a median of 25,500. Reducing program size is therefore essential. Also note, due to this substantial difference in events between completed and uncompleted runs, increasing the timeout yields diminishing returns as it increases the total execution time, while giving confirmation of only a few longer chains. We chose the 1 min timeout as a practical middle ground between unusable runtimes and insufficient coverage.

Takeaway. Hybrid analysis makes whole-kernel analysis feasible: the compiler-based filter allows the model checker to run only on greatly reduced fragments. This avoids timeouts on a majority of dependency chains.

RQ3. How many dependencies are broken by optimizations?

Breaking dependencies. In fact, the absolute number of dependencies *increases* after optimizations, but to validate optimizations, we must ensure that each dependency chain before optimizations, has an equivalent chain after optimizations, as explained in Sec. 5. Fig. 15 shows that for almost half of all dependencies, the compiler-based analysis determines that a dependency is *not* preserved.

Keeping order. Broken dependencies are false positives if the intended order is preserved by other means, which we check through model checking. Order is preserved in 78% of cases. Broken dependencies are reported in 4.4% (result: FAIL). However, after inspecting the generated violation witnesses, they all contain out-of-bounds accesses due to the lack of context for function parameters, e.g., if a function takes a pointer as parameter and the memory region the pointer points to was not properly initialized. A further 16% of chains cause model checker timeouts or errors, leaving their status undecided. Timeouts stem from large chains, while errors arise from an incomplete frontend, especially limited support for `memset` and `memcpy` intrinsics.

8 Related Work

We situate our work in three areas: compiler correctness under weak memory, formal models and tools for the Linux kernel, and dependency-aware compilation.

Compiler correctness. Prior work studies whether compiler transformations preserve semantics via formal verification or testing. CompCert [44] verifies much of a C compiler under sequential semantics, with extensions to relaxed memory [12, 66]. Alive2 [46] uses SMT solving to validate LLVM optimizations. Testing-based approaches detect miscompilations via differential execution and test generation [30, 58]. Téléchat [30] compares source-level litmus tests with assembly-level executions under hardware models, covering the full pipeline, but is limited to small litmus tests. Other work addresses compilation under weak memory, including verified compilation [65], reasoning in Promising Semantics [75], sequential reasoning in concurrent settings [17], and IRs preserving weak-memory behaviors [43]. Vafeiadis et al. [73] show standard optimizations can be unsound under C11, motivating concurrency-aware reasoning.

Hardware memory models and dependencies. Weak hardware models such as ARM [47, 48] and POWER [18, 48] treat dependencies as ordering mechanisms and are formalized in tools such as Armed Cats [5] and HMC [39], with further work on testing weak-memory behaviors [15]. IMM [62] connects language and hardware models, while Compound Memory Models [31] studies heterogeneous compositions.

Software memory models and dependencies. Programming-language memory models differ in their treatment of dependencies. The C11/C++ model [10, 11] and operational

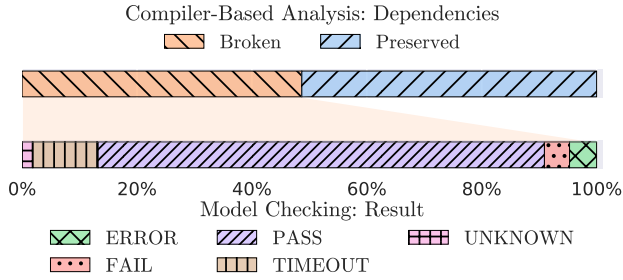


Figure 15. Distribution of dependencies considered broken by the compiler-based analysis, and results of DARTAGNAN after analyzing broken chains.

accounts [59] do not treat dependencies as ordering constraints. RC11 [42], Promising Semantics [37], and subsequent work [57] structure reasoning under relaxed memory. This mismatch with hardware motivates IMM [62]. Related work studies thin-air behaviors [16], dependency tracking [36], robustness [14, 49], and consistency checking [72]. MRD and sMRD [64] incorporate dependencies at the language level, aligning more closely with hardware.

LKMM and tooling. Herding Cats [9] introduced cat [3] and herd for executable axiomatic models. LKMM extends this with explicit dependency ordering [8]. Tools such as herd7 [6] and klitmus analyze litmus tests. Other work studies LKMM complexity and dependency reasoning [22, 34]. Kernel tools like Lockdep [67], KCSAN [24, 25], Sparse [1], and Clang’s `-Wthread-safety` target races, locking, or annotations. Additional work includes probabilistic testing [27] and stateless model checking for RCU [38]. These approaches either operate at litmus-test scale or target bug classes unrelated to dependency ordering. None checks whether compilers preserve dependency chains across the full kernel.

In summary, prior work either verifies transformations under fixed (non-LKMM) semantics, analyzes litmus tests that do not scale to production code, or targets concurrency bugs unrelated to dependency ordering. To our knowledge, this is the first approach combining formal dependency semantics, whole-kernel compiler-based analysis, and targeted bounded model checking to check dependency preservation at scale.

9 Discussion: The Way Forward

For over a decade, the kernel community debated whether compilers silently break dependency ordering, without tooling to answer it. We provide the first answer: under the evaluated compilers and configurations, the kernel’s *semantic dependency chains are intact*.

Our framework enables regression testing for LKMM: model revisions, compiler upgrades, or new optimizations can be validated automatically against the kernel, and thus provide the foundation to close the compiler gap. No com-

piler today treats dependencies as first-class semantic objects, so closing this gap requires a new toolchain.

9.1 Handling of Dependencies in the Linux Kernel

The reasons the kernel survives matter as much as the result. Of the dependencies our compiler-based analysis flags as broken, the majority (78%) are still ordered by other mechanisms—inline assembly preventing optimizations, locks limiting interleavings, or semantics-preserving transformations. This figure is inflated by static-analysis imprecision, but even accounting for that, the kernel’s safety rests on defense-in-depth, not on any single formal guarantee.

Fragile implementation. Many LKMM primitives rely on inline assembly that compilers do not optimize through, incidentally preserving dependencies. This is fragile, as future optimizations may break them without warning.

Documented antipatterns. The kernel documentation lists patterns known to break dependencies; we use several as litmus tests (Sec. 7). In practice, such patterns are typically protected by locks that make the reordering unobservable.

Residual uncertainty. Our model checker flagged 4.4% of checked chains as potentially reordered; inspection of the witnesses shows these are false positives caused by out-of-bounds accesses through uninitialized pointer parameters at the isolated entry function. A further 16% remain undecided due to model checker timeouts or errors. We discuss limitations of our implementation in Sec. 5.6 and Sec. 6.3

Takeaway: Under the evaluated compiler versions, configurations, and given limitations, the Linux kernel is likely not at risk from broken dependencies—but this relies on implementation details, not formal guarantees.

9.2 Characterization of Dependencies

The LKMM must evolve as new code patterns emerge and compiler behavior changes; when its semantics fail to capture real behavior, the model must be revised.

Should `addr` carry dependencies through memory? In a previous revision of the LKMM, `data` dependencies were adjusted to carry through memory [60]. In the same discussion, `addr` dependencies were not adjusted, since common compiler transformations would break them. Later, a separate discussion identified an `addr` dependency in real kernel code that relied exactly on propagation through memory. Our semantics (Sec. 4) already support `addr` propagation through memory, making our analysis applicable to both the current LKMM and potential future revisions.

Regression testing for model evolution. Each LKMM revision requires thorough inspection to avoid mislabeling correct behavior as bugs. Our pipeline can re-analyze the entire kernel after each change, enabling regression testing.

Takeaway: The **LKMM** undergoes continuous revision. Our tool provides the automated, whole-kernel regression testing this evolution demands.

9.3 Toward Dependency-Preserving Compilation

Why compilers do not preserve dependencies today. Language memory models are defined at the source level, but compilers discard this structure during lowering. Frameworks such as MLIR aim to retain high-level semantics but do not yet scale to the kernel. Axiomatic models like **LKMM** require reasoning about executions, which is acceptable for model checkers but too expensive for production compilers.

Not all dependencies require expensive reasoning, however. Identifying a syntactically-checkable subset similar to **Sec. 4** partitions dependencies into syntactically and semantically verifiable ones, analogous to partitioned memory models [74], enabling efficient compile-time checking.

Attempted compiler-side solutions. The compiler community has recognized the need for dependency-aware primitives but has not converged on a viable mechanism. In 2021, Elver proposed a `mustcontrol` attribute for Clang/LLVM [23]. LLVM reviewers were skeptical; counter-proposals such as `__builtin_load_with_control_dependency()` were discussed but never implemented. Separately, McKenney et al. proposed `_Dependent_ptr` [28], a C type qualifier to explicitly mark dependency-carrying pointers. It was never adopted, partly due to the annotation burden on kernel developers. These efforts demonstrate that the problem is well-understood by both communities, but the right abstraction—simple enough for adoption, expressive enough for correctness—has not yet been found.

Beyond C: Rust and the LLVM backend. The compiler gap is an LLVM-level problem, not a language-specific one: passes breaking dependencies in C also affect Rust code, since both Clang and rustc lower to LLVM IR. Rust-for-Linux maps Relaxed atomics to `READ_ONCE()` [26], inheriting the same dependency-carrying assumptions without compiler guarantees. Our LLVM-based analysis applies to Rust kernel modules without modification.

Takeaway: Efficient, general-purpose dependency preservation in compilers remains an open problem. The long-term goal is first-class compiler support for dependencies; our work provides the empirical basis and tooling to validate progress toward that goal.

Acknowledgements

We thank our shepherd and the anonymous reviewers for their helpful comments and suggestions, as well as Paul Heidekrüger for his insights and help. This work was supported in part by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), the Intel Trustworthy Data Center of the Fu-

ture (TDCoF), and Google Research Grants. The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany in the programme of “Souverän. Digital. Vernetzt.”. Joint project 6G-life, project identification number: 16KISK002.

References

- [1] [n. d.]. Sparse Documentation. <https://sparse.docs.kernel.org/en/latest/index.html>.
- [2] 2021. [RFC] LKMM: Add volatile_if(). <https://lore.kernel.org/all/YLn8dzbNwvqrqp5@hirez.programming.kicks-ass.net/>
- [3] Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and Semantics of the Weak Consistency Model Specification Language Cat. <https://doi.org/10.48550/arXiv.1608.07531> arXiv:1608.07531 [cs]
- [4] 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. 2019. Who’s afraid of a big bad optimizing compiler? <https://lwn.net/Articles/793253/>
- [5] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (July 2021), 8:1–8:54. <https://doi.org/10.1145/3458926>
- [6] Jade Alglave and Luc Maranget. 2020. The Herd tool suite. <https://github.com/herd/herdtools7>. Accessed: 2026-03-23.
- [7] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. 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) (ASPLOS ’18). Association for Computing Machinery, New York, NY, USA, 405–418. <https://doi.org/10.1145/3173162.3177156>
- [8] Jade Alglave, Luc Maranget, Alan Stern, and Andrea Parri. 2025. Linux-kernel memory consistency model. <https://github.com/torvalds/linux/blob/master/tools/memory-model/linux-kernel.cat>
- [9] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 1–74.
- [10] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 634–648. <https://doi.org/10.1145/2837614.2837637>
- [11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. [n. d.]. Mathematizing C++ Concurrency. ([n. d.]).
- [12] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 107–127. https://doi.org/10.1007/978-3-642-54833-8_7
- [13] Hans Boehm. 2024. P3475R0: Defang and deprecate memory_order::consume. (2024).
- [14] Soham Chakraborty. 2021. Robustness between Weak Memory Models. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, 173–182. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_26
- [15] Soham Chakraborty, Shankara Narayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2024. How Hard Is Weak-Memory Testing? *Proc. ACM Program. Lang.* 8, POPL (Jan. 2024), 66:1978–66:2009. <https://doi.org/10.1145/3632908>
- [16] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 70:1–70:28. <https://doi.org/10.1145/3290383>

- [17] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. 2022. Sequential Reasoning for Optimizing Compilers under Weak Memory Concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 213–228. <https://doi.org/10.1145/3519939.3523718>
- [18] IBM Corporation. [n.d.]. OpenPOWERFoundation PowerInstructionSetArchitecture. <https://files.openpower.foundation/s/9izgC5Rogi5Ywmm>
- [19] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *TACAS (Lecture Notes in Computer Science, Vol. 12079)*. Springer, 378–382. https://doi.org/10.1007/978-3-030-45237-7_24
- [20] Will Deacon. 2020. Dependency Ordering in the Linux Kernel. <https://lpc.events/event/7/contributions/821/>
- [21] Will Deacon. 2021. The Never-Ending Saga of Control Dependencies. <https://lpc.events/event/11/contributions/973/>
- [22] Will Deacon, Peter Zijlstra, Paul McKenney, and Alglave Jade. 2021. The never-ending saga of control dependencies.
- [23] Marco Elver. 2021. [WIP] Support MustControl conditional control attribute. <https://reviews.lvm.org/D103958> Abandoned.
- [24] Marco Elver, Paul E. McKenney, Dmitry Vyukov, Andrey Konovalov, Alexander Potapenko, Kostya Serebryany, Alan Stern, Andrea Parri, Akira Yokosawa, Peter Zijlstra, Will Deacon, Daniel Lustig, Boqun Feng, Joel Fernandes, Jade Alglave, and Luc Maranget. 2020. Concurrency bugs should fear the big bad data-race detector. *Linux Weekly News (LWN)* (2020). <https://lwn.net/Articles/816850/> Series on the Kernel Concurrency Sanitizer (KCSAN).
- [25] Marco Elver and Dmitry Vyukov. [n.d.]. Kernel Concurrency Sanitizer (KCSAN). <https://google.github.io/kernel-sanitizers/KCSAN.html>
- [26] Boqun Feng. 2026. [PATCH 0/2] Provide Rust atomic helpers over raw pointers. <https://lore.kernel.org/all/20260120115207.55318-1-boqun.feng@gmail.com/>
- [27] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 603–616. <https://doi.org/10.1145/3575693.3575729>
- [28] Akshat Garg, Paul E McKenney, and Ramana Radhakrishnan. 2019. `_Dependent_ptr` to simplify carries a dependency. http://www.rdrop.com/~paulmck/submission/DNNNN%20_Dependent_ptr%20to%20simplify%20carries%20a%20dependency.pdf ISO/IEC JTC1/SC22/WG14 DNNNN.
- [29] Natalia Gavrilenco, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for weak memory models: Relation analysis for compact SMT encodings. In *International Conference on Computer Aided Verification*. Springer, 355–365.
- [30] Luke Geeson and Lee Smith. 2024. Compiler Testing with Relaxed Memory Models. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 334–348.
- [31] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Compound Memory Models: Artifact 7*, PLDI (June 2023), 153:1145–153:1168. <https://doi.org/10.1145/3591267>
- [32] Thomas Haas, René Maseli, Roland Meyer, and Hernán Ponce de León. 2023. Static Analysis of Memory Models for SMT Encodings. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023). <https://doi.org/10.1145/3622855>
- [33] Thomas Haas, Roland Meyer, and Hernán Ponce de León. 2022. CAAT: Consistency as a Theory. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022). <https://doi.org/10.1145/3563292>
- [34] Paul Heidekrüger and M Elver. 2022. Status report: Broken dependency orderings in the Linux kernel.
- [35] David Howells, Paul E McKenney, Will Deacon, and Peter Zijlstra. [n.d.]. LINUX KERNEL MEMORY BARRIERS. <https://github.com/torvalds/linux/blob/master/Documentation/memory-barriers.txt>
- [36] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–30.
- [37] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 175–189. <https://doi.org/10.1145/3009837.3009850>
- [38] Michalis Kokologiannakis and Konstantinos Sagonas. 2019. Stateless model checking of the Linux kernel’s read-copy update (RCU). *International journal on software tools for technology transfer* 21, 3 (2019), 287–306.
- [39] Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 1157–1171. <https://doi.org/10.1145/3373376.3378480>
- [40] Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 479–495.
- [41] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. <https://doi.org/10.1145/3062341.3062352>
- [42] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (Barcelona, Spain) (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. <https://doi.org/10.1145/3062341.3062352>
- [43] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representation. *Proc. ACM Program. Lang.* 7, PLDI (June 2023), 183:1872–183:1895. <https://doi.org/10.1145/3591297>
- [44] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert—a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.
- [45] Nian Liu, Binyu Zang, and Haibo Chen. 2020. No barrier in the road: a comprehensive study and optimization of ARM barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (San Diego, California) (PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 348–361. <https://doi.org/10.1145/3332466.3374535>
- [46] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79.
- [47] Arm Ltd. 2025. *Arm® Architecture Reference Manual for A-profile Architecture*. Technical Report.
- [48] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A tutorial introduction to the ARM and POWER relaxed memory models. (2012).
- [49] Roy Margalit. 2024. Robustness against the C/C++11 Memory Model. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1881–1885. <https://doi.org/10.1145/3650212.3685549>

- [50] P McKenney. 2007. What is RCU, fundamentally? <https://lwn.net/Articles/262464/>
- [51] Paul E McKenney. [n. d.]. ordering. <https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/ordering.txt>
- [52] Paul E McKenney. 2014. [RFC][PATCH 0/5] arch: atomic rework. <https://lwn.net/Articles/588432/>
- [53] Paul E McKenney. 2020. control dependencies. <https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/control-dependencies.txt>
- [54] Paul E McKenney and Mauro Carvalho Chehab. [n. d.]. glossary. <https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/glossary.txt>
- [55] Paul E McKenney and Boqun Feng. [n. d.]. PROPER CARE AND FEEDING OF RETURN VALUES FROM `rcu_dereference()`. https://github.com/torvalds/linux/blob/master/Documentation/RCU/rcu_dereference.rst
- [56] Paul E. McKenney and John D. Slingwine. 1998. Read-Copy Update: Using Execution History to Solve Concurrency Problems. *IEEE Transactions on Parallel and Distributed Systems* (1998). http://www.rdrop.com/users/paulmck/rclock/rclockjrn1_tpbs_mathtype.pdf
- [57] Evgenii Moiseenko, Matteo Meluzzi, Innokentii Meleshchenko, Ivan Kabashnyi, Anton Podkopaev, and Soham Chakraborty. 2025. Relaxed Memory Concurrency Re-executed. *Xmm Model Checker Benchmarks* 9, POPL (Jan. 2025), 72:2149–72:2175. <https://doi.org/10.1145/3704908>
- [58] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. *SIGPLAN Not.* 48, 6 (June 2013), 187–196. <https://doi.org/10.1145/2499370.2491967>
- [59] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, Amsterdam Netherlands, 111–128. <https://doi.org/10.1145/2983990.2983997>
- [60] Jonas Oberhauser and Viktor Vafeiadis. 2023. Linux Commit 9ba7d3b – Make plain accesses carry dependencies. <https://github.com/torvalds/linux/commit/9ba7d3b3b826ef47c1b7b8dbc2d57da868168128>
- [61] Antonio Paolillo, Hernán Ponce de León, Thomas Haas, Diogo Behrens, Rafael Lourenco de Lima Chehab, Ming Fu, and Roland Meyer. 2021. Verifying and Optimizing Compact NUMA-Aware Locks on Weak Memory Models. *CoRR abs/2111.15240* (2021). arXiv:2111.15240 <https://arxiv.org/abs/2111.15240>
- [62] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Mechanized in Coq Proofs for Article: Bridging the Gap between Programming Languages and Hardware Weak Memory Models* 3, POPL (Jan. 2019), 69:1–69:31. <https://doi.org/10.1145/3290382>
- [63] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. <https://doi.org/10.1145/3158107>
- [64] Jay Richards, Daniel Wright, Simon Cooksey, and Mark Batty. 2025. Symbolic MRD: Dynamic Memory, Undefined Behaviour, and Extrinsic Choice. *Proc. ACM Program. Lang.* 9, OOPSLA1 (2025), 1858–1882. <https://doi.org/10.1145/3721089>
- [65] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-Memory Concurrency and Verified Compilation. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, New York, NY, USA, 43–54. <https://doi.org/10.1145/1926385.1926393>
- [66] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (June 2013), 22:1–22:50. <https://doi.org/10.1145/2487241.2487248>
- [67] The Linux Kernel documentation. [n. d.]. Runtime Locking Correctness Validator. <https://docs.kernel.org/locking/lockdep-design.html>
- [68] Linus Torvalds. 2012. Re: Memory corruption due to word sharing. <https://gcc.gnu.org/legacy-ml/gcc/2012-02/msg00027.html>
- [69] Linus Torvalds. 2012. Re: Memory corruption due to word sharing. <https://gcc.gnu.org/legacy-ml/gcc/2012-02/msg00013.html>
- [70] Linus Torvalds. 2019. 'WARNING in tracepoint_probe_register_prio (3)' [2019-08-16 22:57]. https://lore.kernel.org/lkml/CAHk=-wi_KeD1M-_SU_H92vJ-yNkDnAGhAS=RR1yNNGWKW+aA@mail.gmail.com/
- [71] Linus Torvalds. 2021. 'Re: [RFC] LKMM: Add volatile_if()' [2021-06-04 21:27]. [https://lore.kernel.org/all/CAHk=-wgmUbU6XPHz=4NFoLMxH7j_SR-ky4sKzOBrcmkv5A\)jow@mail.gmail.com/](https://lore.kernel.org/all/CAHk=-wgmUbU6XPHz=4NFoLMxH7j_SR-ky4sKzOBrcmkv5A)jow@mail.gmail.com/)
- [72] Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogianis. 2023. Optimal Reads-From Consistency Checking for C11-Style Memory Models. *Artifact for Article "Optimal Reads-From Consistency Checking for C11-Style Memory Models"* 7, PLDI (June 2023), 137:761–137:785. <https://doi.org/10.1145/3591251>
- [73] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 209–220.
- [74] Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned Memory Models for Program Analysis. In *Verification, Model Checking, and Abstract Interpretation*, Ahmed Bouajjani and David Monniaux (Eds.). Springer International Publishing, Cham, 539–558.
- [75] Junpeng Zha, Hongjin Liang, and Xinyu Feng. 2022. Verifying Optimizations of Concurrent Programs in the Promising Semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 903–917. <https://doi.org/10.1145/3519939.3523734>

Appendix

This appendix contains additional details about our semantics, implementation, and evaluation.

Contents

- [Appendix A](#): Artifact description
- [Appendix B](#): Implementation details of our compiler-based analysis and model checking pipeline
- [Appendix C](#): Full semantics of our operational characterization of dependencies
- [Appendix D](#): Algorithmic description of our compiler-based analysis
- [Appendix E](#): Additional evaluation results, including results for Arm64

A Artifact

A.1 Abstract

This appendix helps to reproduce all results reported in [Sec. 7](#). Our artifact includes:

- Annotated Linux Kernel sources
- Modified Clang including our analysis outlined in [Sec. 5](#)
- Modified DARTAGNAN with updated [LKMM ctrl](#) dependencies.

The setup will compile the entire Linux Kernel under the analysis, keeping track how long each compilation and analysis phase takes ([Fig. 13](#)). After compilation, DARTAGNAN will check each flagged pair of dependent accesses. Depending on the chain length (number of events), model checking time increases ([Fig. 14](#)). The final report states how many dependent accesses could be reordered, indicating a miscompilation ([Fig. 15](#)).

We support x86-64 and Arm as target architectures.

A.2 Artifact check-list (meta-information)

- **Programs:** Clang, DARTAGNAN
- **Compilation:** Static analysis
- **Data set:** Linux Kernel sources
- **Hardware:** x86-64 (e.g. AMD EPYC 7713P), Arm (e.g. Ampere-1a)
- **Metrics:** Compile time, Failing model checks
- **How much disk space required (approximately)?:** 50Gb
- **How much time is needed to prepare workflow (approximately)?:** 10min
- **How much time is needed to complete experiments (approximately)?:** 3h
- **Publicly available?:** yes

A.3 Description

A.3.1 How to access.

- Artifact: zenodo.org/records/21899811
- Full Docker images:
 - [rhrssebastian/lkmm-depchecker-aarch64](#)
 - [rhrssebastian/lkmm-depchecker-x86-64](#)

- Linux: [github:TUM-DSE/linux-lkmm-depsan](#)
- Clang: [github:TUM-DSE/llvm-lkmm-depsan](#)

A.3.2 Hardware dependencies. Hardware used for our experiments: x86-64 (AMD EPYC 7713P), Arm (Ampere-1a). Both systems should have enough cores and memory to compile the Linux Kernel in a reasonable amount of time.

A.3.3 Software dependencies. When building from scratch:

- All dependencies needed to build clang (llvm.org/docs/GettingStarted.html)
- All dependencies to build Linux (github.com/torvalds/linux/tree/master)

A.4 Installation

Our artifact can be build either from scratch using the Nix package manager, or be executed from the Docker image we provide for both target architectures.

From source. We recommend the use of the reproducible Nix package manager: nixos.org/download/. The setup requires the feature flakes to be enabled: nixos.wiki/wiki/flakes.

To build the Linux Kernel under our analysis:

```
## Aarch64 or X86_64
# get a shell with all dependencies
# including our modified clang

nix develop

# run the main evaluation script
# Step 1: builds the Kernel config
# Step 2: builds the Kernel
# Step 3: runs DARTAGNAN
# Step 4: produces figures

./scripts/depsan/go.sh
```

From the Docker image. We packaged the binaries to our modified clang together with the Linux Kernel sources in a Docker image. To use it, prepare an output directory for the plots and run:

```
## Aarch64 or X86_64

docker load < <image tarball>

# run the entire analysis
# and write figures to $(pwd)/output

docker run -rm \
  -cpuset-cpus=0-31 \
  -tmpfs /build:rw,exec,size=200g \
  -e OWNER_UID=$(id -u) -e OWNER_GID=$(id -g) \
  -v $(pwd)/output:/output \
  lkmm-depchecker-<arch>:latest 2> /dev/null
```

Note that the timing of the compilation and specifically the analysis is heavily influenced by the write performance of the underlying filesystem. To avoid analysis times from being dominated by writing the results, the Docker image needs a fast `tmpfs` mount as a backend.

A.5 Experiment workflow

Kernel configuration. First, a Kernel configuration is build with realistic, production options (e.g. no sanitizers) and all modules enabled (highest code coverage).

Kernel build. Next, we build the entire kernel and perform our compiler-based analysis at the same time. Depending on the number of cores available this takes around one hour. Per module build, in addition to the actual target code, we produce statistics regarding our analysis and minimal LLVM IR snippets *per function* starting dependency chains.

Model checking. For all dependency chains flagged as broken by the compiler analysis, we invoke DARTAGNAN on the IR snippet. Depending on the number of cores used, this may take several hours.

Plotting. Finally, we produce all plots found in the main paper (x86-64) and appendix (Arm).

A.6 Evaluation and expected results

Compilation and compiler-based analysis. The compilation should succeed, and timing results of our analysis should be similar to our reported numbers. Note that different systems may result in a different distribution. This includes different number of cores, amount of memory (swapping), and the underlying file system.

Model checking. Model checking time should increase linearly with the number of events in the checked chains. The distribution of model checking results should be similar to our reported numbers. Note that different host systems might cause different code to be compiled, which might include inline assembly. Especially on Arm, DARTAGNAN's support for inline assembly is limited and can cause errors.

A.7 Methodology

Submission, reviewing and badging methodology:

- www.acm.org/publications/policies/artifact-review-badging
- cTuning.org/ae/submission-20201122.html
- cTuning.org/ae/reviewing-20201122.html

B Implementation

This section discusses implementation details of the analyses in Sec. 5 and Sec. 6. In particular, we describe problems and limitations we encountered, how we address them, and the implications of our implementation choices.

B.1 Compiler-Based Analysis

We build our compiler-based analysis on top of the LLVM compiler toolchain and use it to verify whether `clang` uses

any optimization on LLVM IR that breaks LKMM dependencies. We implement the analysis in two passes in the standard clang pipeline. One pass performs the analysis on unoptimized LLVM IR, directly after being generated from the AST. The second pass performs the analysis on optimized LLVM IR, directly before lowering to MIR.

Lost language constructs. In order to correctly identify Linux kernel primitives while searching through LLVM IR, we annotate them in the Linux kernel sources, similar to KCSAN [24, 25]. These annotations are the only required changes to the Linux kernel, and they do not affect optimizations or code generation. The changes are contained in 7 header files that define primitives such as `READ_ONCE`, `rcu_dereference`, and `atomics`. Each annotated primitive may expand to multiple IR operations depending on the target and configuration. We treat the resulting volatile accesses or architecture-specific inline assembly as the corresponding marked accesses in LLVM IR.

Control dependencies. Control dependencies as defined in the Linux kernel documentation [51] pose a subtle problem to our analysis: LLVM IR does not have the ability to map instructions back to the language constructs they are generated from. However, Linux kernel `ctrl` dependencies use the scope (“legs”) of the `then` or `else` branch of an `if` statement to construct the set of dependent accesses. Since this syntactical information is not available during compiler-based analysis, we slightly redefine `ctrl` dependencies:

“A control dependency extends from a marked load (`READ_ONCE()` or stronger) through an “if” condition to a marked store (`WRITE_ONCE()` or stronger) that is executed only by one of the legs of that “if” statement that does not *post-dominate* the condition.”

This more accurately captures “a later store’s execution depends on a test of a value computed from a value returned by an earlier load” [54], without relying on a particular syntax.

B.2 Bounded Model Checking

We use the DARTAGNAN bounded model checker [19, 61]. Together with HERD7 [9], DARTAGNAN is the only tool we are aware of that can handle the full LKMM. While HERD7 can only reason about programs written in the litmus format, DARTAGNAN can additionally reason about LLVM IR.

Model checking kernel fragments. Like any other model checker, DARTAGNAN suffers from the state-space explosion problem. The number of possible executions grows with the number of diverging control-flow instructions and thread interleavings. Additionally, the number of possible reorderings depends on the total number of instructions. Despite combining an efficient SMT encoding and several tricks to improve scalability [29, 32, 33], applying DARTAGNAN to the full Linux kernel is infeasible due to the kernel size.

To address this issue, we exploit the observation that most dependencies identified during compiler-based analysis span only a small number of functions. Given a dependency chain produced by the compiler-based analysis, we need a function F from which both the beginning and the end of the chain are reachable. This function serves as the entry point for the model checker. We select F such that after inlining all function calls, the resulting program has minimal code size.

As is common practice in model checking, we assume non-deterministic values for the input parameters of the entry function. Notice that this may lead to false positives if the values chosen by the SMT solver do not correspond to valid inputs in an actual function call.

Reusing compiler-based annotations. While DARTAGNAN's litmus front-end precisely identifies LKMM atomics, its LLVM front-end loses C-level constructs, just like our compiler-based analysis. In order to create the correct events with their corresponding LKMM semantics, we reuse the same annotations for marked accesses and additionally annotate operations coming from fences. Read-modify-write atomics require reconstructing addresses and values from registers in inline assembly, which is not well supported in DARTAGNAN, so we ignore them. As we show in Sec. 7, event ordering in dependency chains is preserved in most cases even without them.

Control dependencies. The cat language supports two notions of **ctrl**: the one used by hardware memory models and the one used by LKMM, with $\emptyset \subseteq \text{ctrl}(lkmm) \subseteq \text{ctrl}(hw)$. For the litmus front-end, DARTAGNAN properly recognizes if statements and computes the post-dominance based notion $\text{ctrl}(lkmm)$. In LLVM IR, however, the code only contains gotos and labels, so DARTAGNAN falls back to the hardware-based notion $\text{ctrl}(hw)$ which may lead to false negatives from the model checker. To overcome this, we ran a second experiment setting $\text{ctrl} = \emptyset$. If the dependency reported by the compiler-based analysis is reported as a false positive with $\text{ctrl} = \emptyset$, the same would be true with the notion of **ctrl** from the kernel. In this setting, the percentage of compiler-based analysis reported chains to be confirmed as a false positive is 71.45% for x86_64 and 40.2% for Arm64.

C Semantics

In Fig. 8 in Sec. 4 we gave the partial operational semantics of a restricted C language with which **data**-, **addr**-, and **ctrl**-dependencies are generated during program execution. Those semantics were partial as they presented only those rules that provided a unique insight. Here we present the complete collection of rules in Fig. 16 for ' $\Downarrow_{\text{dep}}^{\text{stmt}}$ ', Fig. 17 for ' $\Downarrow_{\text{dep}}^{\text{expr}}$ ', and Fig. 18 for ' $\Downarrow_{\text{eval}}$ '. We briefly explain the rules not previously explained in Sec. 4.2.

Statement dependencies. Fig. 16 defines ' $\Downarrow_{\text{dep}}^{\text{stmt}}$ ', which explains how **data**-, **addr**-, and **ctrl**-dependencies are generated when executing **stmts**, as defined in Fig. 7. We explain

several rules here:

- **WHILE-COND/END** first evaluates the condition. If the condition is true (*i.e.* $\neq 0$), it obtains the **data**-, **addr**-, and **ctrl**-dependencies from the body. Crucially, it propagates the data sources (D_c) that affected the branch condition as **ctrl**-sources for **ctrl**-dependencies inside the body. Those **ctrl**-sources *also* propagate to repeating loop executions. Once the condition becomes false (*i.e.* $= 0$), no further dependencies are produced. Crucially, any propagated **ctrl**-sources *do not propagate beyond* the **while**-block – like we illustrated for **if-else** statements in Fig. 6.
- **STORE-U** shows an *unmarked* store. Unlike the marked store (**STORE-M** explained in Sec. 4.2), it does not generate dependencies to itself. However, it *does* propagate dependencies *through memory*, which Fig. 4a illustrated. In particular, those data sources (*i.e.* load_m expressions) affecting the address (D_a) and the value (D_v) are kept in the memory environment M_d . Those will thus propagate to future *marked* stores whose value depends on reading from the respective location in memory. Observe that **addr**- and **ctrl**-dependencies can still be generated whenever the address or data expressions contain load_m expressions. Those dependencies thus target only those marked loads, *not* the outer **store_u**.

Expression dependencies. Fig. 17 defines ' $\Downarrow_{\text{dep}}^{\text{expr}}$ ', which explains how **data** sources, **addr**-, and **ctrl**-dependencies are generated when executing **exprs**. As expressions do not contain stores, they cannot themselves be the target of **data**-dependencies – they do not generate them. However, as they can contain multiple (marked) loads, those loads do serve as *data sources* that appear in the domain of a dependency whenever the expression's result is used. Here we explain several interesting rules of ' $\Downarrow_{\text{dep}}^{\text{expr}}$ ' that were omitted in Sec. 4.2.

- **LOAD-U** defines how an *unmarked* load propagates dependencies. Unlike the *marked* load (in **LOAD-M**), it does not serve as a data source itself, nor does it generate **addr**- and **ctrl**-dependencies to itself. Similar to the marked load, it cannot be the target of **data**-dependencies either. However, observe that data sources, and **addr**- and **ctrl**-dependencies generated by its address expression *do* propagate – this happens when that address expression contains marked loads.
- **ADD** defines how dependencies from separate sub-expression combine under the addition operator; it simply unions the respective sets.

Evaluation. Fig. 18 defines ' $\Downarrow_{\text{eval}}$ ', explaining how expressions reduce to a value. Its definitions are entirely standard, where **LOAD-M/U** load a value from memory M , while **VAR** retrieves a value from the local environment Γ . We define these rules for completion, as they affect control flow and thus the generation of dependencies.

$\frac{\langle \langle \Gamma, M \rangle, e \rangle \Downarrow_{\text{eval}} v \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D, a, c \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{assign } n e \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma[n \mapsto v], M, \Gamma_d[n \mapsto D], M_d \rangle, \langle \emptyset, a, c \rangle \rangle} \text{ ASSIGN}$
$\frac{\langle \langle \Gamma, M \rangle, e_c \rangle \Downarrow_{\text{eval}} v_c \quad v_c \neq 0 \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_c \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_c, a_c, c_c \rangle \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \cup D_c \rangle, s_{\top} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_s, a_s, c_s \rangle \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{if } e_c s_{\top} s_{\perp} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_s, a_c \cup a_s, c_c \cup c_s \rangle \rangle} \text{ IF-TRUE}$
$\frac{\langle \langle \Gamma, M \rangle, e_c \rangle \Downarrow_{\text{eval}} 0 \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_c \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_c, a_c, c_c \rangle \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \cup D_c \rangle, s_{\perp} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_s, a_s, c_s \rangle \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{if } e_c s_{\top} s_{\perp} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_s, a_c \cup a_s, c_c \cup c_s \rangle \rangle} \text{ IF-FALSE}$
$\frac{\langle \langle \Gamma, M \rangle, e_c \rangle \Downarrow_{\text{eval}} v_c \quad v_c \neq 0 \quad \langle \langle \Gamma, M, \Gamma_d, M_d \rangle, e_c \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_c, a_c, c_c \rangle \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \cup D_c \rangle, s \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_1, a_1, c_1 \rangle \rangle \quad \langle \langle \Gamma', M', \Gamma'_d, M'_d, S_c \cup D_c \rangle, \text{while } e_c s \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma'', M'', \Gamma''_d, M''_d \rangle, \langle d_2, a_2, c_2 \rangle \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{while } e_c s \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma'', M'', \Gamma''_d, M''_d \rangle, \langle d_1 \cup d_2, a_c \cup a_1 \cup a_2, c_c \cup c_1 \cup c_2 \rangle \rangle} \text{ WHILE-COND}$
$\frac{\langle \langle \Gamma, M \rangle, e_c \rangle \Downarrow_{\text{eval}} 0}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{while } e_c s \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma, M, \Gamma_d, M_d \rangle, \langle \emptyset, \emptyset, \emptyset \rangle \rangle} \text{ WHILE-END}$
$\frac{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, s_0 \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma', M', \Gamma'_d, M'_d \rangle, \langle d_1, a_1, c_1 \rangle \rangle \quad \langle \langle \Gamma', M', \Gamma'_d, M'_d, S_c \rangle, s_1 \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma'', M'', \Gamma''_d, M''_d \rangle, \langle d_2, a_2, c_2 \rangle \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{concat } s_0 s_1 \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma'', M'', \Gamma''_d, M''_d \rangle, \langle d_1 \cup d_2, a_1 \cup a_2, c_1 \cup c_2 \rangle \rangle} \text{ CONCAT}$
$\frac{\langle \langle \Gamma, M \rangle, e_a \rangle \Downarrow_{\text{eval}} a \quad \langle \langle \Gamma, M \rangle, e_v \rangle \Downarrow_{\text{eval}} v \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_a \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_a, a_a, c_a \rangle \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_v \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_v, a_v, c_v \rangle \quad d_m = \{ \langle d, \text{id} \rangle \mid d \in D_v \} \quad a_m = \{ \langle a, \text{id} \rangle \mid a \in D_a \} \quad c_m = \{ \langle c, \text{id} \rangle \mid c \in S_c \}}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, (\text{store_m } e_a e_v) @ \text{id} \rangle \Downarrow_{\text{dep}}^{\text{stmt}} \langle \langle \Gamma, M[a \mapsto v], \Gamma_d, M_d[a \mapsto D_a \cup D_v] \rangle, \langle d_m, a_v \cup a_a \cup a_m, c_v \cup c_a \cup c_m \rangle \rangle} \text{ STORE-M}$
$\frac{\langle \langle \Gamma, M \rangle, e_a \rangle \Downarrow_{\text{eval}} a \quad \langle \langle \Gamma, M \rangle, e_v \rangle \Downarrow_{\text{eval}} v \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_a \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_a, a_a, c_a \rangle \quad \langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, e_v \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_v, a_v, c_v \rangle}{\langle \langle \Gamma, M, \Gamma_d, M_d, S_c \rangle, \text{store_u } e_a e_v \rangle \Downarrow_{\text{exec}} \langle \langle \Gamma, M[a \mapsto v], \Gamma_d, M_d[a \mapsto D_a \cup D_v] \rangle, \langle \emptyset, a_v, c_v \rangle \rangle} \text{ STORE-U}$

Figure 16. Semantics ' $\Downarrow_{\text{dep}}^{\text{stmt}}$ ', defining how **addr**-, **data**-, and **ctrl**-dependencies are generated for **stmts** (from Fig. 7).

$$\begin{array}{c}
\frac{}{\langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, \text{const } v \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle \emptyset, \emptyset, \emptyset \rangle} \text{CONST} \qquad \frac{}{\langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, \text{var } n \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle \Gamma_d[n], \emptyset, \emptyset \rangle} \text{VAR} \\
\\
\frac{\langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, e_x \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_x, a_x, c_x \rangle \quad \langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, e_y \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_y, a_y, c_y \rangle}{\langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, \text{add } e_x e_y \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_x \cup D_y, a_x \cup a_y, c_x \cup c_y \rangle} \text{ADD} \\
\\
\frac{\langle \langle \Gamma, \mathbf{M} \rangle, e_a \rangle \Downarrow_{\text{eval}} a \quad \langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, e_a \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_a, a_a, c_a \rangle \quad a_m = \{ \langle d_a, \text{id} \rangle \mid d_a \in D_a \} \quad c_m = \{ \langle s_c, \text{id} \rangle \mid s_c \in S_c \}}{\langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, (\text{load_m } e_a) @ \text{id} \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle \mathbf{M}_d[a] \cup D_a \cup \{\text{id}\}, a_a \cup a_m, c_a \cup c_m \rangle} \text{LOAD-M} \\
\\
\frac{\langle \langle \Gamma, \mathbf{M} \rangle, e_a \rangle \Downarrow_{\text{eval}} a \quad \langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, e_a \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle D_a, a_a, c_a \rangle}{\langle \langle \Gamma, \mathbf{M}, \Gamma_d, \mathbf{M}_d, \mathbf{S}_c \rangle, \text{load_u } e_a \rangle \Downarrow_{\text{dep}}^{\text{expr}} \langle \mathbf{M}_d[a] \cup D_a, a_a, c_a \rangle} \text{LOAD-U}
\end{array}$$

Figure 17. Semantics ‘ $\Downarrow_{\text{dep}}^{\text{expr}}$ ’ defining how *data sources*, *addr*-, and *ctrl*-dependencies are generated for *exprs* (from Fig. 7).

$$\begin{array}{c}
\frac{}{\langle \langle \Gamma, \mathbf{M} \rangle, \text{const } v \rangle \Downarrow_{\text{eval}} v} \text{EV-CONST} \qquad \frac{}{\langle \langle \Gamma, \mathbf{M} \rangle, \text{var } n \rangle \Downarrow_{\text{eval}} \Gamma[n]} \text{EV-VAR} \\
\\
\frac{\langle \langle \Gamma, \mathbf{M} \rangle, e_x \rangle \Downarrow_{\text{eval}} x \quad \langle \langle \Gamma, \mathbf{M} \rangle, e_y \rangle \Downarrow_{\text{eval}} y}{\langle \langle \Gamma, \mathbf{M} \rangle, \text{add } e_x e_y \rangle \Downarrow_{\text{eval}} x + y} \text{EV-ADD} \\
\\
\frac{\langle \langle \Gamma, \mathbf{M} \rangle, e_v \rangle \Downarrow_{\text{eval}} v}{\langle \langle \Gamma, \mathbf{M} \rangle, \text{load_m } e_v \rangle \Downarrow_{\text{eval}} \mathbf{M}[v]} \text{EV-LOAD-M} \qquad \frac{\langle \langle \Gamma, \mathbf{M} \rangle, e_v \rangle \Downarrow_{\text{eval}} v}{\langle \langle \Gamma, \mathbf{M} \rangle, \text{load_u } e_v \rangle \Downarrow_{\text{eval}} \mathbf{M}[v]} \text{EV-LOAD-U}
\end{array}$$

Figure 18. Semantics ‘ $\Downarrow_{\text{eval}}$ ’ defining how *exprs* (from Fig. 7) reduce to values.

```

1 %tmp = alloca i32
2 ...
3 %0 = load volatile i32, ptr @X !READ_ONCE
4 store i32 %0, ptr %tmp !READ_ONCE
5 %a = load i32, ptr %tmp !READ_ONCE
6 ...
7 call void asm "" ... !SMP_MB
8 ...
9 %10 = icmp eq i32 %a, 0
10 br i1 %10, label %THEN, label %ELSE
11 EN:
12 store volatile i32 1, ptr @Y !WRITE_ONCE
13 br label %END
14 SE:
15 ...

```

Figure 19. Unoptimized LLVM IR including annotations

D Algorithmic Implementation

This section outlines an algorithmic description of the compiler-based analysis for detecting broken dependencies, as described in Sec. 5.

To track the origins of values used in marked accesses we heavily utilize LLVMs use-def chains. Since the analysis concerns *syntactic* dependencies, the core of the algorithm is a depth-first search (DFS) following the use-def chains, regardless of the operations performed on them.

Reverse search. The compiler-based analysis performs the DFS in reverse, starting from operands of a marked access and following use-def chains backward to identify the source of the dependency. The access and operand that starts the search define the kind of dependency:

- Address dependencies start at the address operand of a marked load or store.
- Data dependencies start at the value operand of a marked store.
- Control dependencies start at the condition of a branch instruction.

Fig. 19 shows an example of unoptimized LLVM IR of the program shown in Fig. 12a including metadata added through kernel annotations. Note again, a single marked access in C results in multiple LLVM instructions. We define the single volatile access or inline assembly call for each marked C access as the defining access in LLVM IR.

Algorithm 1 Identify Dependency Search Starting Points

```

Procedure FINDSTARTINGPOINTS(program)
  # addr dependencies
  for each marked store s in program do
    SEARCH(s.address, [])
  end for
  for each marked load l in program do
    SEARCH(l.address, [])
  end for
  # data dependencies
  for each marked store s in program do
    SEARCH(s.value, [])
  end for
  # ctrl dependencies
  for each branch b in program do
    SEARCH(b.condition, [])
  end for

```

Algorithm 2 performs the depth-first search, traversing backwards through operands extending the dependency chain *s* passed as the second argument, until reaching a marked load or encountering a termination condition. The **TERMINATES** predicate determines whether an LLVM value (usually the result of an operation) should terminate the current search. Examples of terminating values include constants, returns from external functions or intrinsics, and pointers from `alloca` instructions. To implement the dependency propagation through memory described in Sec. 4, the search continues after any load, if a po-earlier store to the same SSA address-value exists.

Algorithm 2 Depth-first Dependency Search

```

Procedure SEARCH(v, s)
  if !TERMINATES(v) then
    if v.type is load then
      SEARCHSTORES(v.address, s + [v])
    end if
    if v.type is marked load then
      RECORDDEPENDENCY(s + [v])
    end if
    for op ∈ v.operands do
      SEARCH(op, s + [v])
    end for
  end if

```

RECORDDEPENDENCY then simply stores the identified dependency chain for later comparison for **addr** and **data** dependencies. For **ctrl** dependencies, it is still necessary to identify po-later marked accesses that *do not* post-dominate the branch *b* currently heading the (reverse) chain *s*. We again leverage existing LLVM infrastructure, which provides a post-dominator tree.

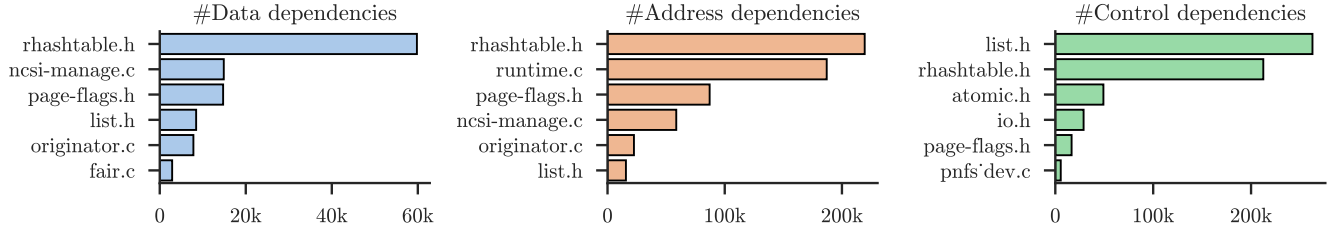


Figure 20. Six most common files starting dependency chains. (x86_64)

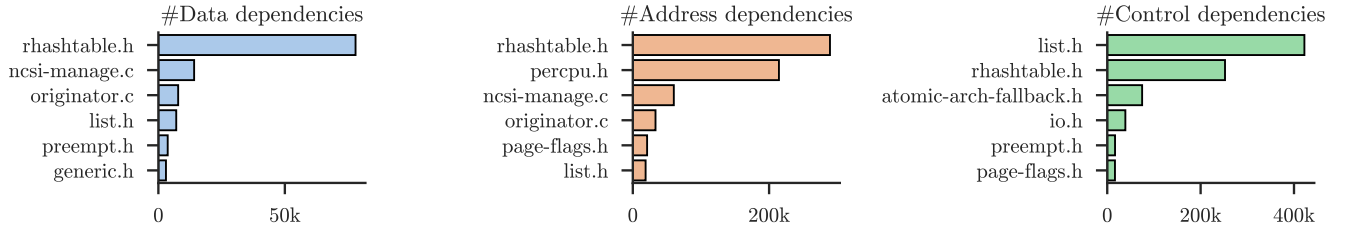


Figure 21. Six most common files starting dependency chains. (Arm64)

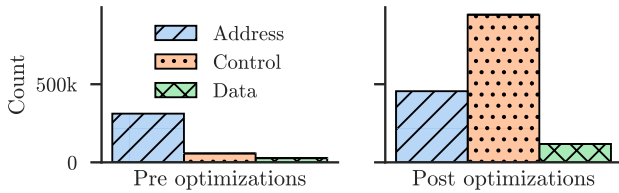


Figure 23. Total number of dependencies found by static analysis. Before and after optimizations. (Arm64)

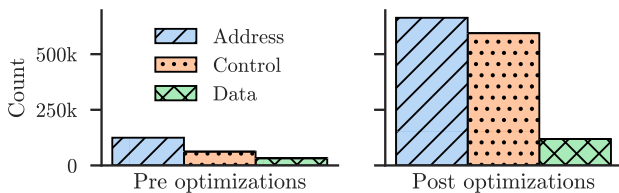


Figure 22. Total number of dependencies found by static analysis. Before and after optimizations. (x86_64)

E Additional Evaluation Results

E.1 Continuation of Sec. 7 (x86_64)

Total number of dependencies. Notably, the number of detected dependencies increases by 6.8× after optimization. Passes such as `instcombine` can break dependencies by replacing dependent values with non-dependent ones, but they can also create new dependencies. Loop unrolling has a particularly strong effect on `ctrl` dependencies because the static analysis does not reason about loops: if an unrolled body contains a marked access that can end a dependency, the analysis will report one chain per unrolled copy, all guarded by the same branch.

Fig. 20 shows the six files starting the most dependencies, before and after optimizations combined. Aside the headers defining data structures and primitives discussed in the main paper, heavy users of dependencies are low-level modules, such as `page-flags.h` and `ncsi-manage.c`.

E.2 Evaluation on Arm64

The results on Arm64 are similar to x86_64, with a few implementation-caused differences.

Compiler-based analysis. Fig. 24 shows the time of each step in our compiler-based analysis on the Arm64 machine. Compiling the full Linux kernel takes only 180 minutes (machine dependent), of which 30.3% is spent in middle-end optimizations. Again, the analysis takes up most of the optimization runtime (70%). Pre- and post-optimization passes each account for 44% of the analysis time, with the remaining 12% spent on dependency matching.

Model checking. Fig. 25 again shows that DARTAGNAN’s execution time grows roughly linearly with the number of events in program fragments.

Breaking dependencies. Fig. 26 shows the share of broken and preserved pre-optimization dependencies on Arm64. Notably, the compiler-based analysis determines that 71% of dependencies are *not* preserved, as opposed to only 50% on x86_64.

Observing that the compiler-based analysis generally finds less `add` dependencies after optimizations on Arm64 (Fig. 23), we attribute that to architecture specific code breaking our implementation, requiring further investigation.

Keeping order. Fig. 26 shows the results of DARTAGNAN with respect to Arm64: 52% of checked chains result in an

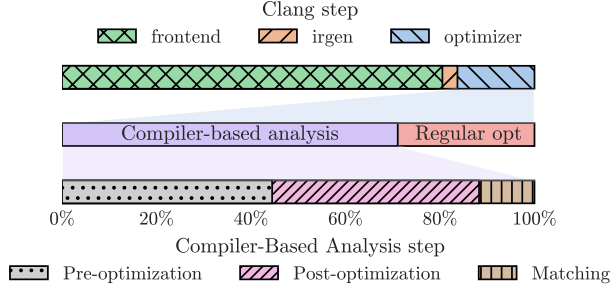


Figure 24. Distribution of the time for compiler-based analysis as a fraction of the total compile time. (Arm64)

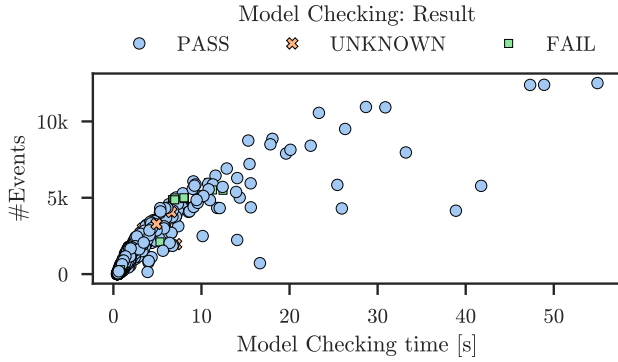


Figure 25. Execution time of DARTAGNAN per chain, relative to the number of events in the analyzed chain. (Arm64)

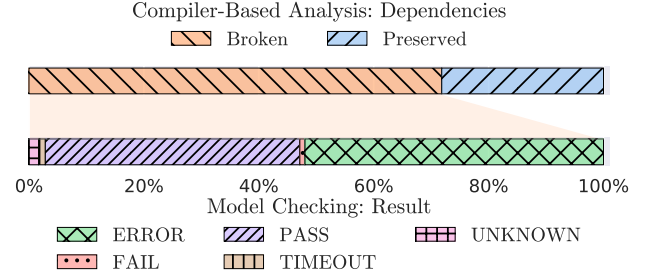


Figure 26. Distribution of dependencies considered broken by the compiler-based analysis, and results of DARTAGNAN after analyzing broken chains. (Arm64)

error. The main reasons are the presence of `blockaddress` operations coming from usages of *labels as values* in the kernel and `memset` and `memcpy` intrinsics. DARTAGNAN reports that 44.29% of the chains from the compiler-based analysis are false positives.