Understanding Atomics and Memory Ordering Issues in Real-World Rust Software

Cheng Wang[1,2,3], Tengfei Tu*[1,2], Sujuan Qin[1,2], Guangjun Wu[3], Fei Gao[1,2], Mingchao Wan[4]
[1]State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications
[2]School of Cyberspace Security, Beijing University of Posts and Telecommunications
[3]Zhongguancun Laboratory [4]Beijing Academy of Blockchain and Edge Computing
{wangc2023, tutengfei.kevin, qsujuan}@bupt.edu.cn, wugj@zgclab.edu.cn, gaof@bupt.edu.cn, chainmaker@baec.org.cn

Abstract—**Rust is designed as a systems programming language that aims to provide safety guarantees and performance efficiency. In practice, programmers usually use atomic correlations to share data across threads. For example, by using atomic operations to correlate with non-atomic addresses, they can design lock-free data structures for efficient concurrency. Although atomic operations are used in safe code, memory ordering misuses can still lead to atomic concurrency bugs and performance loss.**

In this paper, we conduct the first empirical study of atomic operations and memory ordering usage in Rust, manual inspection of 2883 atomic usages in real-world applications, including 15 thread bugs and 150 performance issues. We also study their usage scenarios, performance comparisons and issue fixes to provide a better understanding on Rust's memory ordering misuses and guide better code practices in the future.

We design `AtomVChecker`, an automated static analyzer to detect memory ordering misuses. we evaluate our tool on four widely-used concurrent libraries, it can automatically analyze 228 atomic correlations with 80% accuracy. Based on the atomic correlation analysis, `AtomVChecker` finds a total of 51 performance loss issues in 9 Rust packages, with all of them recently confirmed by the project maintainer based on our reports.

Index Terms—**Software and its engineering, Automated static analysis, Software safety**

I. Introduction

Rust [1] is a systems programming language that focuses on performance, and security. Over the past few years, Rust has gained substantial popularity [6]–[8], particularly in developing system-level applications like operating systems and performance-critical databases [9]–[12]. Because of its performance and safety advantages, Rust has clinched the title of "Most Loved Programming Language" in Stack Overflow Developer Survey for seven years [2]–[8].

Concurrency bugs have always been a hot topic and a challenging area, these bugs are difficult to reproduce and are widely studied [13]–[15]. However, previous work has not delved into concurrency bugs and performance issues caused by atomic operations and memory ordering usage in Rust real-world applications. As a result, programmers still face considerable confusion about atomics and memory ordering [16]–[19], leading to concurrency safety and performance loss in real-world applications [20]–[22].

*Corresponding author.

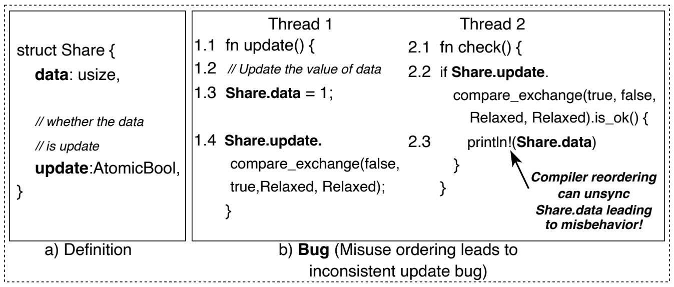
Our research is based on a novel observation that atomic operations can synchronize with other non-atomic addresses, which require memory ordering to ensure synchronicity. For ease of description, we refer to the non-atomic addresses as atomic correlated addresses, and the synchronicity among them as atomic correlations.

Figure 1 gives an example to demonstrate atomics and memory ordering issues. There is an atomic correlation between *data* and *update*. Thread 1 writes to *data*, then marks *update* as true, and *data* must use `Acquire`/`Release` to synchronize with *update* when thread 2 reads *update* as true. However, incorrect use of `Relaxed` allow the compiler and processor to reorder the execution of line 1.3 and line 1.4. The atomic concurrency bug occurs if the execution of line 1.4 and line 1.3 in thread 1 is interleaved by the execution of line 2.2 and line 2.3 in thread 2, line 2.3 will output an unupdated *data*. Although it is correct to use `SeqCst`, incorrect use of `SeqCst` can cause performance loss, especially for underlying applications. Overall, memory ordering misuses risk concurrency bugs and performance loss, the Rust community [23], [24] highlights their importance.

In this paper, we present the first empirical study of practices and issues related to atomics and memory ordering in Rust real-world applications. Our study pays particular attention to how the five types of memory ordering affect concurrency safety and performance loss. Based on our research findings, we present AtomVChecker, a static analyzer that can automatically analyze atomic correlations to detect memory ordering misuses. We validate its effectiveness in real-world applications, identifying 51 unknown memory ordering misuses.

Overall, the paper makes the following contributions:

- We conduct the first Rust empirical study of memory ordering usage, analyzing 2883 cases of atomic usage and evaluating the performance gap between different memory orderings.
- We summarize 4 insights and 4 suggestions about memory ordering through in-depth analysis of real-world applications, which will guide future research in system development, testing, and error detection.
- We design AtomVChecker, a static analyzer that can detect error-prone parts of memory ordering and provide appropriate recommendations for memory ordering, helping developers correct potential errors.



```
struct Share {
    data: usize,

    // whether the data
    // is update
    update:AtomicBool,
}
```

a) Definition

```
                Thread 1                         Thread 2
1.1  fn update() {                    2.1  fn check() {
1.2   // Update the value of data     2.2  if Share.update.
1.3   Share.data = 1;                        compare_exchange(true, false,
                                               Relaxed, Relaxed).is_ok() {
1.4   Share.update.                   2.3      println!(Share.data)
        compare_exchange(false,              }
        true,Relaxed, Relaxed);            }
      }
```

Compiler reordering can unsync Share.data leading to misbehavior!

b) **Bug** (Misuse ordering leads to inconsistent update bug)

Fig. 1. Sample code for misuse of memory ordering

- We define atomic correlations to capture common patterns of memory ordering misuses. This approach is suitable for inferring such issues and reveales 51 unknown memory ordering misuses.

Availability: Our research artifacts, source code, and datasets are publicly available at https://github.com/AtomVChecker/AtomVChecker.

II. Background

In this section, we provide the background knowledge necessary to understand the rest of this paper. We begin with an introduction to the concurrency basics of Rust, and then elaborate on atomics and the instruction reordering strategies for various memory orderings.

A. Concurrency basics of Rust

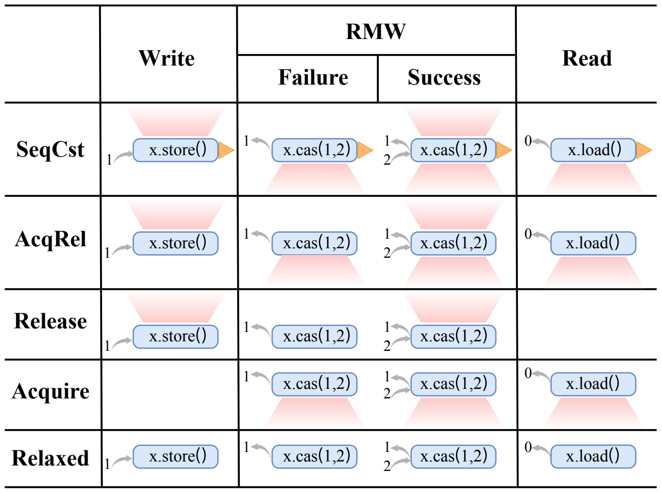
Rust provides several thread synchronization primitives for safe and efficient concurrency [25], with message passing and shared-memory synchronization as the predominant methods. Similar to Go, Rust provides `channel` for message passing, the two ends are the `Sender` and the `Receiver`. The `Sender` is responsible for sending data from one thread and the `Receiver` receives data in another thread.

Shared-memory synchronization provides traditional synchronization primitives, including lock (`Mutex`), conditional variable (`Condvar`), and atomic read/write (`atomic`). Similar to C++, Rust's std::sync::atomic module provides a set of atomic operations, including `load`, `store`, `fetch_add`, `compare_exchange`, whose indivisible nature ensures that the result of the operation can only be observed upon its completion, while the process remains uninterrupted [26]. For ease of description, we call `fetch_add`, `compare_exchange` and other atomic operations that involve both read and write operations as RMW operations.

B. Memory model of Rust

Rust memory model inherites from C++ but there are some differences [23], [27], [28]. The most significant difference is the removal of `Consume` in Rust. Although `Consume` is lighter than `Acquire`, no compiler actually implements `Consume` because it is difficult to understand and verify [23], which can easily lead to concurrency safety issues in practice.

Each atomic operation takes a std::sync::atomic::Ordering indicating the strength of the memory barrier. Compilers and



	Write	RMW		Read
		Failure	Success	
SeqCst	1 x.store()	1 x.cas(1,2)	1 2 x.cas(1,2)	0 x.load()
AcqRel	1 x.store()	1 x.cas(1,2)	1 2 x.cas(1,2)	0 x.load()
Release	1 x.store()	1 x.cas(1,2)	1 2 x.cas(1,2)	
Acquire		1 x.cas(1,2)	1 2 x.cas(1,2)	0 x.load()
Relaxed	1 x.store()	1 x.cas(1,2)	1 2 x.cas(1,2)	0 x.load()

TABLE I
Composition and Role of Memory Ordering

processors may reorder memory accesses during the actual program execution to improve performance. The role of memory barrier is to specify how memory accesses should be ordered around an atomic operation to ensure concurrency safety. Rust provides a set of five enumerated memory orderings, each with specific implications for program behavior. To elucidate their roles, Table I introduces unique symbols for a more comprehensive explanation.

Values read by atomic loads or RMWs are indicated by upper arrows, and values written by atomic stores or RMWs are indicated by lower arrows. The upper trapezoid symbolizes `Release` constraint, which prevents prior memory accesses in current thread from reordering past it. The lower trapezoid represents `Acquire` constraint, which prevents subsequent memory accesses in current thread from reordering before it. AcqRel-loads behave like Acquire-loads, AcqRel-stores behave like Release-stores, while AcqRel-RMWs simultaneously exhibit both `Acquire` and `Release` properties. The triangle on the right signifies `SeqCst` constraint, which provides stronger constraints than `Release`/`Acquire` and `AcqRel`. `Release`/`Acquire` and `AcqRel` guarantee the reordering constraints within specific threads, while `SeqCst` provides global consistency guarantee, meaning that all threads observe all modifications in the same order [29]. `Relaxed` does not impose any constraints, and only ensures atomicity and modification order consistency [30].

III. Methodology

This section discusses how we analyze and collect memory ordering misuses, including concurrency bugs and performance loss. Then, we design a static analyzer to detect memory ordering misuses.

Exploring Software and Libraries. Our project selection criteria include open source, active maintenance, and a long code history. Following these criteria, we select seven representative real-world applications, including three underlying systems (*SnarkOS* [9], *Occlum* [31], *Servo* [32]), one secure

Application	Start Time	Stars	Commits	LOC
tikv	2015/12	14076	7624	571K
teaclave	2017/04	1116	649	163K
sled	2016/01	7429	4373	25K
servo	2012/02	24675	47511	9473K
influxdb	2013/09	27107	49161	272K
occlum	2018/11	1232	1356	89K
snarkOS	2020/02	3062	8739	23K
libraries	2014/10	11146	3653	48K

TABLE II
INFORMATION OF SELECTED APPLICATIONS. THE START TIME, NUMBER OF STARS, COMMITS, AND TOTAL SOURCE LINES OF CODE.

Software	Total	Atomic Usage						
		Scenarios Usage			Ordering Usage			
		Struct	Global	Other	Relaxed	Rel/Acq	AcqRel	SeqCst
tikv	836	72.37%	5.74%	21.89%	32.36%	12.97%	2.69%	51.98%
teaclave	206	64.08%	20.39%	15.53%	33.93%	10.27%	1.34%	54.46%
sled	211	78.20%	13.27%	8.53%	27.98%	46.33%	1.84%	23.85%
servo	189	71.96%	19.58%	8.46%	46.07%	10.99%	1.05%	41.89%
influxdb	138	74.64%	1.45%	23.91%	62.94%	4.19%	0.70%	32.17%
Occlum	70	81.43%	12.86%	5.71%	24.67%	33.76%	3.90%	37.67%
SnarkOS	33	84.85%	3.03%	12.12%	39.39%	0%	0%	60.61%
libraries	1200	76.00%	8.67%	15.33%	41.82%	32.07%	5.58%	20.53%

TABLE III
ATOMIC USAGE IN REAL-WORLD RUST PROGRAMS.

computing platform (*teaclave* [33]), and three storage systems (*sled* [11], *tikv* [12], *influxdb* [10]), all using atomics extensively for efficient concurrency. Additionally, we explore six widely-used Rust libraries (*Rayon* [36], *Parking-Lot* [34], *Tokio* [35], *Actix* [37], *Crossbeam* [38], *Once_cell* [39]).

Studying memory ordering misuses. To understand atomics and memory ordering issues, we explore from two orthogonal dimensions: incorrect use of weak memory orderings can cause concurrency bugs, and incorrect use of strong memory orderings can cause performance loss.

For the first dimension, we filter GitHub commit logs of the applications listed in Table II and online vulnerability datasets(CVE [40] and RustSec [41]), using keywords such as ”atomic” and ”ordering” to identify concurrency bugs. Then, we study patches and online discussions of each bug to delve into its origins and behaviors.

For the second dimension, we carry out comparative experiments on specific processor architecture to evaluate the performance loss associated with different memory orderings in different levels of concurrency.

Detecting memory ordering misuses. To detect memory ordering misuses, we define 6 atomic correlation patterns and corresponding memory ordering requirements(§V-C). As we will show, memory ordering misuses often manifest themselves between atomic operations. Accordingly, we propose 8 combinations of atomic correlation pairs on top of Rust MIR(§V-D), and use data flow analysis [42], data dependency [43] and control dependency analysis [43] to detect atomic concurrency bugs(§IV-B) and performance loss(§IV-C).

Threats to validity. Threats to the validity of this study come from many sources. We focus on representative Rust applications, noting that our study may not cover all memory ordering misuses outside our study's scope. Due to the randomness of instruction reordering in different architectures, it is difficult to reproduce these issues. Despite these limitations, we diligently collect and analyze such issues, and design static analyzer to detect them. We believe that our work can help programmers understand memory ordering and use our detector to ensure the security of atomics and memory ordering.

IV. ATOMIC USAGES

This section presents our findings on atomics and memory ordering issues, including the reasons for atomic usage, the root causes and solutions for atomic concurrency bugs, and practical quantification of performance loss.

A. Reasons for usage

To understand how and why programmers use atomic types, we first study how atomic types are used in real-world Rust applications. One of the design goals of atomics in Rust is to provide efficient concurrent processing capabilities. Therefore, we pose the question: how ”efficient” are atomics compared to other concurrency primitives? To answer this question, we conduct simple tests to evaluate the performance difference between atomic and other concurrency primitives. Our experiments show that the concurrent counter with atomic::fetch_add is 5-6 times faster than mutex and 3-4 times faster than channel. In some complex scenarios, the difference in performance may be more noticeable.

To delve deeper into the reasons why programmers use atomic types, we collect a total of 2883 atomic operations, as shown in Table III. Specifically, atomic types are often used in concurrent data structures to ensure efficient concurrency, with their usage ranging from 64.08% to 84.85%. Additionally, they are used in global static variables ranging from 1.45% to 20.39%(e.g., the global counter). Other cases include verifying code correctness in test code, and facilitating the mutual conversion between atomic type references and mutable references. For example, in Occlum, AtomicU32::from_mut() is used to get atomic access to &mut U32.

An interesting observation from Table III is that libraries use `SeqCst` at an average rate of 20.53%, while a significant proportion of the real-world applications rely heavily on `SeqCst`, with usage rates ranging from 23.85% to 60.61%. We further randomly select 150 SeqCst uses. 80% of them use `SeqCst` without any comments, and manually adjusting to a weak memory ordering with expert knowledge does not cause compilation errors. For 16% of these usages, `SeqCst` is used as a warning for complex logic. Only 4% of these usages actually require `SeqCst` for specific concurrency needs. We also notice that the Rust community is increasingly discussing the misuse of `SeqCst`, and the code has been optimized accordingly [44]–[46].

Observation 1: In real-world applications, programmers prefer to use atomic operations with the strongest memory ordering, `SeqCst`, to ensure safe concurrency.

Insight 1: There are many cases where SeqCst is not necessary, which is not good code practices and can be optimized with a weak memory ordering.

B. Atomic concurrency bugs

Compilers and processors implement various tricks to increase program efficiency, but these optimizations ignore multi-thread interactions [23]. Atomic concurrency bugs occur in concurrent code when atomic correlated addresses fail to synchronize with atomic operations with weak memory orderings that leads to data races.

We study atomic concurrency bugs from the first dimension of memory ordering misuses, collecting a total of 15 atomic concurrency bugs caused by incorrect use of Relaxed. Notably, while atomic operations are considered safe by the Rust compiler, we find 14 of the studied atomic concurrency bugs occur within the safe code. We conduct a detailed analysis of the collected bugs and identify two bug patterns : *critical-state inconsistent update bug(CIU)* and *atomic related Concurrency bug(ARC)*.

Critical-state inconsistent update bug(CIU). Atomic operations can correlate with the critical state of non-atomic memory addresses. In weakly ordered architectures [48], such as ARM64, atomic correlation violations occur when atomic operations with weak memory orderings in concurrent code cause other threads to fail to synchronize the changes to these critical states, which can lead to *critical-state inconsistent update bug*.

Figure 2(a)(b) shows a real-world bug in Crossbeam's safe code. This bug violates the atomic correlation between *ready* and *data*, where *ready* is used to correlate the state of *data*. As shown in the figure, thread 1 updates *data* and uses **Relaxed-store** to set *ready* to true without any constraints, compilers and processors could reorder the two operations. If the reordering happens, `pop()` will return an uninitialized *data* when the execution of **Relaxed-store** to *ready* and update to *data* of thread 1 is interleaved by the execution of **Relaxed-load** to *ready* and read to *data* in thread 2 . This bug can be fixed by setting the load/store of *ready* to Acquire/Release to ensure synchronization between them.

AtomicPtr related Concurrency bug(ARC). Such bugs manifest only in AtomicPtr and are architecture-specific, such as DEC Alpha that lacks data dependency [49], [50]. AtomicPtr operations target the addresses, which mean that the corresponding content operations are not atomic. Therefore, AtomicPtr inherently establishes an atomic correlation between the atomic pointer and the content. Atomic correlation violations occur when atomic operations with weak memory orderings in concurrent code cause other threads to fail to synchronize the content pointed by the atomic pointer, which can result in *atomicPtr related Concurrency bug*.

Figure 2(c)(d) shows a real-world bug in Thread_local [47]'s safe code that violates atomic correlation between AtomicPtr and the corresponding content. In thread 2, **Relaxed-load** to *bucket* lacks a data dependency, there is no happens-before relationship with Dereferencing *bucket*. If the reordering happens, it will access a uninitialized content of *bucket* when the execution of Dereferencing *bucket* and **Relaxed-load** to *bucket* in thread 2 is interleaved by the execution of initializes the content and **Relaxed-store** *bucket_ptr* to *bucket* in thread 1. This bug can be fixed by setting the load/store of bucket to Acquire/Release to ensure data dependency barrier.

Observation 2: On specific architectures, there are also atomic concurrency bugs from safe code that has passed the Rust compiler's safety checks.

Observation 3: Most atomic concurrency bugs can be fixed in a simple way (modifying the memory ordering) and these fixes are related to the cause of bugs.

insight 2: Incorrect use of `Relaxed` by programmers is the main reason for atomic concurrency bugs in safe code, then it is better marked as unsafe.

Insight 3: The notable link between causes and their simple fixes suggests the feasibility of creating an automated tool to detect and fix atomic concurrency bugs.

C. Atomic performance issues

We examine atomic performance gap from the second dimension of memory ordering misuses, designing multiple producer and consumer scenarios to evaluate the performance loss of different memory orderings. Specifically, we choose Crossbeam-skiplist, which makes heavy use of Relaxed in its code, as the subject of our experiments. Fine-tune the memory ordering while ensuring program correctness (Relaxed to AcqRel/Release, SeqCst), and repeat our experiment at different concurrency levels (10^1, 10^2, 10^3, 10^4, 10^5) to evaluate the average elapsed time.

We run the experiments on a machine with 16GB RAM and 8 core of CPU. Figure 3 shows that when concurrency levels are low (e.g., 10^1, 10^2), the performance gap between different memory orderings is not significant: 1.62% to 2.59% between SeqCst and Relaxed, 0.68% to 1.51% between Acquire/Release and Relaxed, and 0.93% to 1.06% between SeqCst and Acquire/Release. However, at high concurrency levels (e.g., 10^3, 10^4, 10^5), the performance gap between memory orders become more pronounced: 7.94% to 12.54% between SeqCst and Relaxed, 5.03% to 7.18% between Acquire/Release and Relaxed, and 2.77% to 5.00% between SeqCst and Acquire/Release.

Observation 4: In high concurrency scenarios, data-engaging multi-core processor environments, the choice of memory ordering has a large impact on program performance, but a small impact on program performance in low concurrency scenarios.

Insight 4: In low concurrency scenarios, choosing SeqCst streamlines coding and prioritizes business logic. In high concurrency scenarios, however, it is better to choose the correct memory ordering to ensure program efficiency and security.

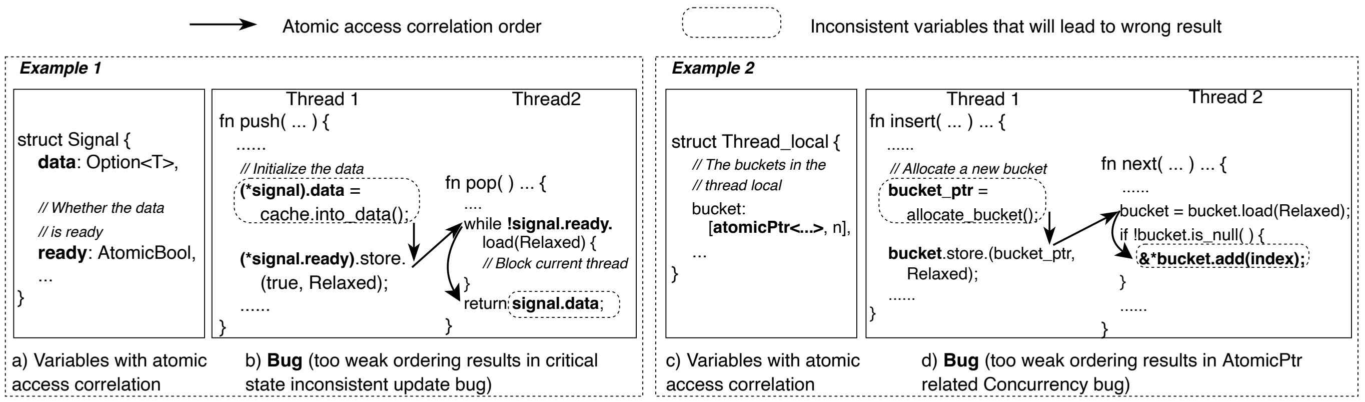


Fig. 2. Two bugs from Crossbeam and Thread_local that violate atomic correlation.

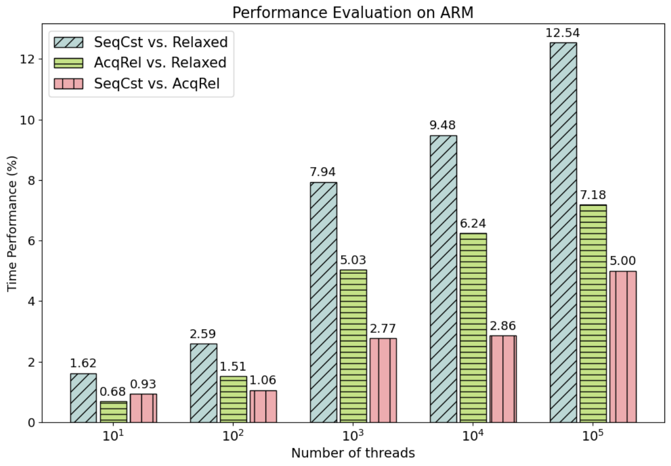


Fig. 3. Performance gap for memory ordering

V. Approach

In this section, we describe the method for defining atomic correlations and analyzing atomic correlation violations to identify memory ordering misuses.

A. Design

We design AtomVChecker, a static analyzer based on Rust MIR to analyze and detect potential memory ordering misuses in Rust applications. MIR is an intermediate representation of the compilation process that basically consists of two components: *local* and *basic block*. *local* represents stack-allocated memory address, including user-defined variables and temporaries, which allows to track data flow across the program and reflects the order of local address allocations necessary for AtomVChecker. *basic block* comprises a series of statements and a terminator. Each statement is an expression composed of multiple locals that is used to represent specific memory operations. The terminator is a key determinant of the control flow, indicating how the basic block transitions to other basic blocks.

AtomVChecker models atomic operations based on MIR and quickly analyzes atomic correlated locations, then uses data flow analysis, control dependencies, and data dependencies automatically to detect memory ordering misuses. Figure 4 outlines the comprehensive framework. Overall, it contains four layers: 1) Input; 2) Preprocess; 3) Static Analysis; 4) Detection. The input is a Rust project and user-supplied analysis options, and the output is a warning about potential memory ordering misuses and related code snippets with recommendations for ordering usage.

B. Preprocess

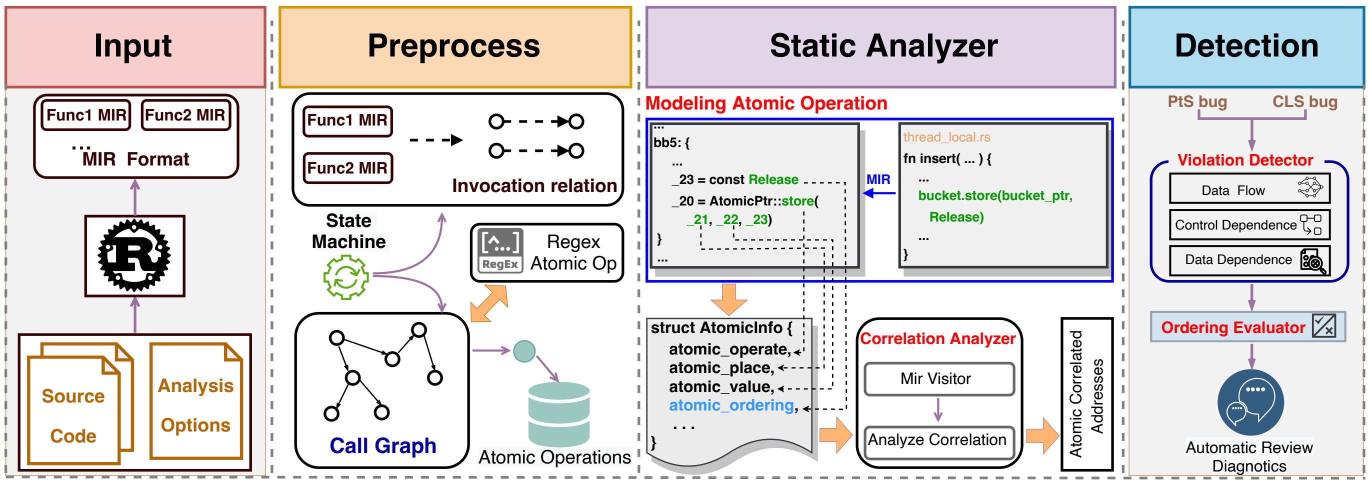
1) Building Call Graph: After the Rust compiler generates all instance functions of the program, AtomVChecker builds the call graph through a finite state machine (FSM). As illustrated in Figure 5, FSM consists of 6 states: START, INST, VISITOR, CALLEE, EDGE, and END, where START is the initial state of the FSM and END is the final state. Transitions ①–⑤ specify the prerequisites for state changes, where each instance method acts as a trigger event for FSM.

We illustrate how FSM works with Figure 2(a)(b). After obtaining the instance method *push*, FSM converts to INST. Then, we get the MIR form of *push* and FSM converts to VISITOR. The method *store* is called in *push*, we get the corresponding callsite and FSM converts from VISITOR to CALLEE. FSM converts to EDGE when we add a directed edge from *push* to *store* in the call graph, with the callsite as its weight. FSM repeats these steps until it processes the final instance method. Finally, AtomVChecker generates a call graph that maps instance methods to nodes and callsites as the weights for its edges.

TABLE IV
Matching rules in AtomVChecker. Atomicisize*, Atomicusize*: includes different types for different scenarios.

	Libraries	Types	Methods
Atomic Operation	std, core	AtomicBool, Atomicisize*, Atomicusize*, AtomicPtr	load, store, CompareExchange, FetchUpdate

2) Identify atomic operations: AtomVChecker gets the *DefPath* for all methods, which defines the path from the root crate to the method, such as the *Atomicbool-load* in `std`, the corresponding DefPath is `std::sync::atomicbool::load`.



```
...
bb5: {
  ...
  _23 = const Release
  _20 = AtomicPtr::store(
    _21, _22, _23)
}
...
```

```
fn insert( ... ) {
  ...
  bucket.store(bucket_ptr,
    Release)
  ...
}
```

```
struct AtomicInfo {
    atomic_operate,
    atomic_place,
    atomic_value,
    atomic_ordering,
    ...
}
```

Fig. 4. The architecture of AtomVChecker.

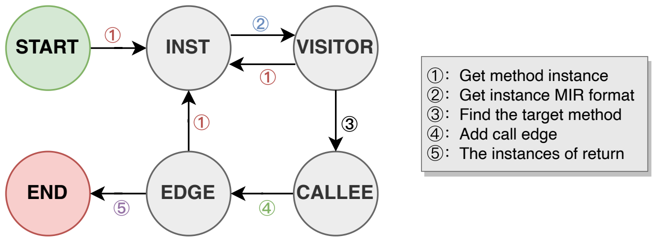

START
INST
VISITOR
CALLEE
EDGE
END
①
②
①
③
①
④
⑤
①: Get method instance
②: Get instance MIR format
③: Find the target method
④: Add call edge
⑤: The instances of return

Fig. 5. Parsing Instance Method Invocation.

Then, based on Table IV, AtomVChecker uses regular expressions to identify atomic operations by matching specific **methods** of **types** in the **libraries** with ”Atomic” prefix, and passes them to Static Analysis stage for further analysis.

C. Static Analysis

1) Atomic Correlation Patterns: Atomic correlation allows atomic operations to synchronize non-atomic addresses, which is the key to lock-free concurrent programming. Combining observations from the use of real-world atomic operations, we enumerate several classical patterns of atomic correlation and ordering requirements (minimum) as shown in Table V. Concisely, for two variables x and y, we use the following notion to represent an atomic correlation:

$$O_1(x) \Rightarrow O_2(y)$$

where O_1 and O_2 can represent atomic or non-atomic operations. For example, **atomic-load(x)** $\Rightarrow$ **read(y)** described in Pattern 1 means that an atomic-load to x semantically requires a subsequent non-atomic-read to y, `Acquire` is required to ensure that the state of x must be checked or read before reading y; Similarly, **write(x)** $\Rightarrow$ **atomic-store(y)** described in Pattern 2 means that an atomic-store to y semantically requires a prior non-atomic-write to x, `Release` is required to ensure that the non-atomic write to x is completed before updating y.

Now, we formally define the atomic concurrency bugs(see Table VI). Bugs often appear as pairs of the atomic correlation

TABLE V
SIX PATTERNS WITH ATOMIC CORRELATION AND CORRESPONDING REQUIREMENT OF MEMORY ORDERING. MEMORY ORDERING*:MINIMUM MEMORY ORDERING REQUIREMENT.

Pattern No.	Atomic Correlation	Definition	Memory Ordering*
1	atomic(x).load $\Rightarrow$ read(y)	Each load to x semantically requires a subsequent read to y	Acquire
2	write(x) $\Rightarrow$ atomic(y).store	Each store to y semantically requires a prior write to x	Release
3	write(x) $\Rightarrow$ atomic(y).RMW	Each RMW to y semantically requires a prior write to x	Release
4	atomic(x). RMW $\Rightarrow$ read(y)	Each RMW to x semantically requires a subsequent read to y	Acquire
5	write(x) $\Rightarrow$ atomic(y).RMW, atomic(y).RMW $\Rightarrow$ read(x)	Each RMW to y semantically requires a prior write to x and a subsequent read to x	AcqRel
6	None	Atomic operation does not correlate with any variable	Relaxed

across threads, but using a weak memory ordering violates the atomic correlation, manifesting as CIU bugs or ARC bugs. For example, figure 2(a)(b) corresponds to the Case 1, where writing to *signal.data* will bring update to *signal.ready*, and reading *signal.data* must check the status of *signal.ready*, which requires Acquire/Release to synchronize atomic correlated address(*signal.data*). Incorrect use of `Relaxed` violates atomic correlation, leading to a CIU bug.

2) Transforming atomics into Design Pattern: In order to eliminate the complexity of retrieving and processing atomic operations during Detection, we propose the following definitions related to atomic correlation violation:

Definition 1: (AtomI). AtomI use a 6-tuple to denotes an atomic operation: $\langle Operate, Place, Value, Ordering, Caller, Location\rangle$, where $Operate$ is the type of atomic operation, including Load, Store and RMW. $Place$ denotes the stack address corresponding to the atomic operation, $Value$ denotes the target stack address of the atomic operation, specially, for RMW, $Value$ includes the target addresses of both atomic read and write operations. $Ordering$ denotes the

TABLE VI
CASES OF ATOMIC CORRELATION PAIRS. THERE ARE 8 TYPES OF CORRELATION PAIRS. HERE WE SHOW ONLY 2 TYPES FOR ILLUSTRATION, THE REMAINING SIX TYPES ARE PATTERN 1 && PATTERN 3, PATTERN 4 && PATTERN 2, PATTERN 1 && PATTERN 5, PATTERN 2 && PATTERN 5, PATTERN 3 && PATTERN 5 AND PATTERN 4 && PATTERN 5.

Case No.	Pattern Pair	Explaination	Example
1	Pattern 1 && Pattern 2	Each load and store requires Acquire/Release to synchronize non-atomic variables	Figure 2
2	Pattern 3 && Pattern 4	Each atomic RMW requires Acquire/Release to synchronize non-atomic variables	Figure 1

	Ordering Influence	InterVal
Load	atomic.load	Struct fields and Atomicptr contents within 30% following AtomI.Value.
Store	atomic.store	Struct fields and Atomicptr contents within 30% preceding AtomI.Value.
RMW	atomic.RMW	Struct fields and Atomicptr contents within 30% before and after AtomI.Value.

TABLE VII
INTERVAL FOR DIFFERENT ATOMIC OPERATIONS.

memory ordering of the atomic operation and $Caller$ denotes the instance function that performs the atomic operation. The last element represents the information about the atomic operation at the source code.

Definition 2: (InterVal). InterVal is a list of stack addresses that represents candidate pool of atomic correlated addresses. The numeric size corresponding to the address measures the allocation order of the stack. Theoretically, all addresses can associate with atomic operations, but they often locate in structure fields and the contents pointed by atomic pointers. They also tend to be close to the address of the atomic operation for ease of maintenance. Therefore, AtomVChecker analyzes InterVal from structure fields or contents pointed by atomic pointers within 30% of the atomic operation context.

Definition 3: (Partner). Given an atomic operation, *AtomI* = $\langle$*Operate, Place, $Value$, Ordering, Caller, Location*$\rangle$, and the corresponding $InterVal$, Partner is $\langle$*AtomI, InterVal*$\rangle$, which denotes the atomic correlated addresses corresponding to the atomic operation.

Definition 4: (Influence). For two Partners of an atomic variable:

$Partner$ = $\langle$AtomI, InterVal$\rangle$,

$Partner'$ = $\langle$AtomI$'$, InterVal$'\rangle$,

The Influence is $\langle \text{AtomI}, x, \text{AtomI}', y\rangle$, where x is the atomic correlated address in $InterVal$ and y is the atomic correlated address in $InterVal'$.

Table VI gives the definition to atomic correlation pairs. For example, for *atomic-load* and *atomic-store* on the same address, *atomic-load* is denoted as $AtomI$ whose atomic correlated address is $x(x \in InterVal)$ and *atomic-store* is denoted as $AtomI'$ whose atomic correlated address is $y(y \in InterVal')$. By analyzing the tuple $\langle$AtomI, x, AtomI$'$, $y\rangle$, we can determine whether it matches the atomic correlation pair as described in Case 1.

3) Analyzing atomic correlated addresses: After identifying all atomic operations by regular expression, AtomVChecker extracts `AtomI` from the MIR expressions of atomic operations according to Definition 1. As shown in Figure 4, the MIR form of the *atomic-store* for *bucket* is `_20 = AtomicPtr::store(_21, _22, _23)`, where `_21` is $Place$, `_22` is $Value$, the memory ordering represented by `_23` is $Ordering$, and $Caller$ is the instance method *insert*.

Then AtomVChecker analyzes `InterVal` according to Definition 2. According to Table I, memory ordering affects the scope of optimization reordering, atomic correlated addresses are more likely to occur within the scope affected by memory ordering. Therefore we refine the range of correlated addresses for different atomic operations as shown in Table VII. For RMW that involves both atomic load and store semantics, we treat atomic correlated addresses with load and store as InterVal.

D. Detection

1) Inferring atomic correlation pairs: After analyzing atomic correlated addresses, according to Definition 3, AtomVChecker obtains `Partner` for all atomic operations. Algorithm 1 illustrates how AtomVChecker infers the atomic correlation pairs in Table VI, which consists of three steps:

Step 1: Algorithm 1 traverses each `Partner` in *AtomPs* (line 3). For load and RMW's read semantic of AtomicPtr, **DerefAddr** gets the addresses from `InterVal` that perform dereference operations (line 9). For load and RMW's read semantic of non-AtomicPtr, **GeneratePdg** builds the program dependency graph (pdg) for each $AtomI.Caller$ (line 11). **CtlDep** analyzes the pdg to identify the addresses from `InterVal` that are control dependent on `AtomI` (line 12). If $LoadCorAddrs$ is null, indicating that the atomic operation does not correlate with other addresses, AtomVChecker adds the atomic operation with `Relaxed` to $AtomCors$(line 18).

Step 2: If $LoadCorAddrs$ is not null (line 13), Algorithm 1 processes the remaining `Partner` in *AtomPs* (line 21). It uses **Site** to check if two addresses point at the same atomic variable and identify all write-related `Partner` (store and write semantics of RMW) as $StoreCorrPartners$ (line 23-24). Then, **CorInfer** combines each $Partner'\langle AtomI', InterVal'\rangle$ from $StoreCorrPs$ with $AtomI$ and $LoadCorAddrs$ to generate a sequence of $Influence\langle AtomI, x, AtomI', y\rangle$ for analyzing the atomic correlation pair(line 28), where x comes from the $LoadCorAddrs$ and y comes from $InterVal'$.

Step 3: For non-AtomicPtr, if **site** checks that x and y point at the same varibale, or for AtomicPtr, if **InitPtr** confirms that there is an initialization to the content (line 31), it satisfies the conditions of atomic correlation pair, and adds the atomic operations along with its minimum memory ordering requirements to $AtomCor$(line 29 and 31).

As an example, for Figure2(a)(b), *atomic(signal.ready).load* is control dependent on *signal.data*, there is an atomic correlation described by Pattern 1 listed in Table 5(Step 1). $InterVal$ for *atomic(signal.ready).store* contains *signal.data*, `Influence` is represented as ⟨*ready.load::*$AtomI$, *signal.data*, *ready.store::*$AtomI'$, *signal.data*⟩ (step 2). The correlated addresses in `Influence` point at the same field *signal.data*, therefore, there is an atomic correlation as described by Pattern 2 in Table 5, which satisfies the atomic correlation pair condition of Case 1 in Table 6 (Step 3).

Algorithm 1: Inferring atomic correlation

```
Input: AtomPs: a set of atomic Partners
Output: AtomCors: a set of atomic correlation
1  AtomCors ← {};
2  foreach Partner ∈ AtomPs do
3      AtomI, InterVal ← Partner.getElems();
4      if AtomI.Operate is load or AtomI.Operate is RMW then
5          AtomProcess(AtomI, InterVal, AtomPs, AtomCors);
6  Function AtomProcess(AtomI, InterVal, Atoms, AtomCors):
7      LoadCorAddrs ← {}, LoadAddr ← AtomI.Place;
8      if OpType(AtomI.place) is AtomicPtr then
9          LoadCorAddrs ← DerefAddr(InterVal);
10     else
11         pdg ← GeneratePdg(AtomI.Caller);
12         LoadCorAddrs ← CtlDep(LoadAddr, InterVal, pdg);
13     if LoadCorAddrs then
14         StoreCorrPs ← StoreCorrParts(LoadAddr, AtomPs);
15         AtomCor ← AccCor(StoreCorrPs, AtomI, LoadCorAddrs);
16         add AtomCor to AtomCors;
17     else
18         add {AtomI, Relaxed} to AtomCors;
19 Function StoreCorrParts(LoadAddr, AtomPs):
20     StoreCorrPs ← {};
21     foreach Partner ∈ AtomPs do
22         AtomI ← Partner.AtomI, StoreAddr ← AtomI.Place;
23         if Site(LoadAddr, StoreAddr) and (AtomI.Operate = store
             or AtomI.Operate = RMW) then
24             add Partner to StoreCorrPs;
25     return StoreCorrPs;
26 Function AccCor(StoreCorrPs, AtomI, LoadCorAddrs):
27     AtomCor ← {};
28     Influences ← CorInfer(StoreCorrPs, AtomI, LoadCorAddrs);
29     foreach Influence ∈ Influences do
30         AtomI, x, AtomI′, y ← Influence;
31         if Site(x, y) or InitPtr(y) then
32             add {AtomI, Acquire}, {AtomI′, Release} to
                 AtomCor;
33         else
34             add {AtomI, Acquire}, {AtomI′, Relaxed} to
                 AtomCor;
35     return AtomCor;
```

2) Correlation violation evaluator: After inferring atomic correlation pairs, $CorAtoms$ composes a set of $AtomI$ with corresponding minimum memory ordering requirements. If $AtomI.Ordering$ is too weak, AtomVChecker warns about possible atomic concurrency bugs, and if it is too strong, AtomVChecker warns about possible performance loss. AtomVChecker analyzes the location of the atomic operation with $AtomI.Location$ and provides sensible memory ordering suggestions. In particular, for an $RMW'AtomI$, if both `Acquire` and `Release` are present in the corresponding memory ordering parameter set, AtomVChecker considers the RMW operation to satisfy the atomic correlation of Pattern 5 in Table 5, where the minimum memory ordering requirement should be `AcqRel`.

Library	# Atomic Correlations	# Involved Structures	% False Positive	Analysis Time
Once_Cell	11	4	9.09%	23s
Parking-lot	34	6	17.64%	1m25s
Rayon	21	8	14.29%	7m20s
Crossbeam	162	26	22.84%	35m23s
Total	**228**	**44**	**20.61%***	**44m31s**

TABLE VIII
ATOMIC CORRELATIONS ANALYZED BY ATOMVCHECKER. THE ATOMIC CORRELATIONS PRESENTED HERE ARE THE SIX PATTERNS LISTED IN TABLE 5. *: THE WEIGHTED AVERAGE FALSE POSITIVE RATE.

VI. EVALUATION

In this section, we explore the following research questions to evaluate AtomVChecker.

RQ1: How effective is AtomVChecker in conducting a detailed analysis of atomic correlation?

RQ2: Can AtomVChecker help programmers warn of memory ordering misuse in real-world programs?

RQ3: How does it compare to existing approaches?

To answer RQ1, we select four concurrent libraries from Table II for evaluation that make extensive use of atomic operations. To answer RQ2, we collect a dataset with 2 atomic concurrency bugs from RustSec and 53 Rust crates from GitHub, using AtomVChecker to measure its applicability. To answer RQ3, we compare it to three existing approaches. All the experiments are done on an 20.04-Ubuntu system with a 3.20 GHz Intel processor.

A. Effectiveness of AtomVChecker

To demonstrate the effectiveness of AtomVChecker, we perform AtomVChecker towards the applications listed in Table II to testify whether our design is effective to analyze atomic correlations, which allows accurate identification of memory ordering misuses. Given space limitations, we present only four applications here. For more details, please visit our GitHub repository[1]. As shown in Table VIII, 228 atomic correlations are analyzed, involving 44 concurrent data structures. The analysis is also efficient, AtomVChecker takes 44 minutes to complete the detection of these four libraries.

To evaluate AtomVChecker's accuracy in analyzing atomic correlations, each alert is reviewed by at least two members of our team. We also consult with the crate maintainers based on our understanding. The results show the false positive rate is around 20%.

[1]https://github.com/AtomVChecker/rust-atomic-study/tree/main/section-5-detection

TABLE IX
EXPERIMENT RESULTS OVERVIEW. VIOLATION TYPES INCLUDE CIU(CRITICAL-STATE INCONSISTENT UPDATE BUG), ARC(ATOMICPTR RELATED CONCURRENCY BUG), SMO(PERFORMANCE LOSS CAUSED BY INCORRECT USE OF STRONG MEMORY ORDERING). VIOLITAON REPORT IS PRESENTED AS TRUE POSITIVE/FALSE POSITIVE (- MEANS TP/FP IS 0/0), WE ALSO ANALYZE CODE VOLUME AND DOWNLOAD NUMBERS TO REPRESENT THE EXTENT OF THEIR INFLUENCE, AND * INDICATES INAPPLICABILITY.

Crate	**# Violation Report (TP/FP)**							**Analysis Time (s)**	**Memory Usage (MB)**	**# of Lines**	**All-time Downloads**
	CIU	**ARC**	**SMO**	**Total**	**lockbud**	**Miri**	**loom**				
RS2022-0006	-	1/0	-	1/0	-	1/0	1/0	2	67.34	746	31,643,325
RS2022-0029	1/0	-	-	1/0	-	1/0	1/0	9	368.82	2529	123,629,705
Bottlerocket	-	-	4/0	4/0	-	-	-	679	1131.02	125644	401
Ripgrep	-	-	8/2	8/2	-	-	-	106	379.14	33729	592,968
Termcolor	-	-	2/0	2/0	-	-	-	1	76.55	2095	138,227,682
Coldsnap	-	-	2/0	2/0	-	-	-	842	5076.46	1816	371,646
Fragile	-	-	2/0	2/0	-	-	-	4	45.68	1079	21,434,995
Rust-mysql-simple	-	-	3/0	3/0	-	-	-	66	885.68	8373	1,450,865
Rayon	-	-	2/3	2/3	-	-	-	440	1254.25	117726	96,889,462
Snarkos	-	-	13/3	13/3	-	-	-	824	5765.54	54553	10,007
Occlum	-	-	16/4	16/4	-	-	-	1422	5483.41	81707	2,301
Total	1/0	1/0	51/10	53/10	-	2/0	2/0	4395	*	*	*

B. Applicability of AtomVChecker

To demonstrate the applicability of AtomVChecker, we collect 53 crates that use atomic operations from GitHub, along with 2 atomic concurrency bugs from RustSec. After using AtomVChecker to detect atomic correlation violations within this dataset, it generates 79[2] warnings as shown in table IX. Manual inspection of these warnings shows that AtomVChecker can detect known bugs, and also identifies 51 unknown memory ordering issues in 9 Rust crates.

We also measure the execution time and memory usage of AtomVChecker. AtomVChecker completes all analyzes in 1.2 hours with at most 4.79 GB memory consumption. Overall, AtomVChecker can analyze real-world Rust crates with reasonable computational resources.

C. Comparison with Other Approaches.

Comparison with lockbud. Lockbud [51] includes an algorithm for detecting atomicity violations in Rust programs. However, Lockbud fails to detect the 51 memory ordering misuses found by AtomVChecker. After confirmation with Lockbud's authors, the primary reason is that it has a limited ability to detect atomicity violations and does not consider the effect of memory ordering on concurrent behavior.

Comparison with Miri. Miri [52] is a dynamic analysis tool that detects specific undefined behaviors. We apply Miri to our dataset, it detects two atomic concurrency bugs, but fails to detect the performance loss caused by incorrect use of strong memory ordering. This suggests that AtomVChecker complements Miri in detecting memory ordering misuses and providing memory ordering recommendations.

Comparison with loom. Loom [53] is a model checking tool that can rewrite test cases with loom's atomic types to detect concurrency bugs. The result shows that loom can not find any of the 51 memory ordering misuses found by AtomVChecker because it only focuses on concurrency correctness without considering memory ordering's impact on performance. In addition, its effectiveness largely depends on the test cases written with Loom.

[2]Redundant warnings arise from repeated calls to the same atomic operations in the code path.

VII. DISCUSSION

False positive of atomic correlation analysis. As shown in Table VIII, the atomic correlations analysis still has a 20.61% false positive rate and they come from three main reasons:

(1) Atomic correlation exceptional cases. Atomic operations sometimes correlate with the underlying resource. For example, in parking-lot, *WordLock.state* simulates the state of a lock. `Acquire`/`Release` ensures safe execution of updates on critical sections correlated with atomic operations

(2) Cross-function atomic correlation. Although AtomVChecker's intra-procedural atomic correlation analysis covers most cases, some atomic correlations fall outside its scope. This is especially true if the address read by the atomic pointer is passed or returned to another function.

(3) Inherent Limitations. AtomVChecker cannot detect atomic correlations required by `SeqCst` because it cannot dynamically track memory states across threads to infer the sequential consistency, often misclassifying them as `Acquire`/`Release` or `AcqRel`.

Among the three sources of false positives, the second is the easiest to address. It can be pruned by inter-procedural correlation analysis. Issue 1 emerged as the primary cause of false positives in our experiments. Addressing this issue requires to analyze atomic correlations within specific cases, which is challenging and remains a focus of our future work.

False positive of correlation violation evaluator. As shown in Table IX, the atomic correlation violation also generates some false positives. They come from two main sources: (1) Wrong atomic correlations. All atomic correlations analyzed by AtomVChecker are used directly to infer atomic correlation violations, so wrong atomic correlations cause most of the false positives. (2) Benign atomic correlation violation. Like benign single-thread atomic correlation violations also exist due to special semantics. Issue 2 can be pruned by combining dynamic thread information.

False negatives of atomic correlation analysis and violation detection. AtomVChecker is not infallible. False negatives of atomic correlation violation are caused by atomic correlation analysis missing some true atomic correlations. For the false negatives of atomic correlations, the main reason is that the Rust compiler does not generate code for generic functions unless they are monomorphic, which means that if the generic function is not used, AtomVChecker cannot analyze the MIR to detect memory ordering misuses.

VIII. Related Work

A. Existing works on Rust Static Analysis

Static analysis is a method of analyzing code to discover potential issues without running the code. One of the core components of Rust is borrow checker [54], which performs a static analysis of the ownership and borrowing rules to detect issues like dangling pointers and memory leaks.

Besides the built-in static analyzer, researchers have also used IR to develop static analysis tools for specific issues. LLVM IR serves as a common intermediate language across multiple programming languages (C/C++, Rust), which allows to apply previous LLVM-based detection tools to Rust [55]–[57]. The Rust HIR and MIR are also used to develop static analyzer for detecting thread and memory safety issues. MIRAI [58] is an abstract interpreter of MIR used to check the security of the program. Rudra [59] uses HIR and MIR to detecting common undefined behaviors in Rust programs. SafeDrop [60] detects memory corruptions in Rust programs based on Rust Mir. MIRCHECKER [61] uses MIR to track numeric and symbolic information to detect runtime crashes and memory safety issues.

B. Empirical studies on memory model

Many researchers have previously studied the memory model. In C++, CDSCHECKER [62] uses model checking to simulate different execution paths of memory operations in concurrent code and looks for potential issues in these simulated executions. New model checking tools have been developed to efficiently examine the code of the C/C++ memory model [63]–[65]. There are also tools that can help programmers understand the behaviors of memory orderings [66], [67].

In Rust, loom [53] is designed to detect issues that may arise in Rust concurrent code, especially the parts involving memory model. However, loom requires specific types and operations, making it difficult to adapt directly to real-world applications. Miri [52] is a experimental interpreter for Rust MIR, designed to execute binaries, run cargo project tests, and detect specific undefined behaviors. It only detects bugs when test cases trigger problematic executions and is still experimental in detecting memory ordering misuses.

Our research is different from the previous work. It outperforms model checkers in recognizing atomic correlations to minimize state exploration and prevent state space explosion. Unlike Miri, which relies on specific test cases, AtomVChecker's comprehensive compile-time analysis provides enhanced detection capabilities.

IX. Conclusion

In this paper, we present the first comprehensive empirical study of atomic usage, atomic concurrency bugs and performance loss in real-world Rust applications. Many useful observations and insights are provided in our study. We also design a static analyzer, AtomVChecker, to detect memory ordering misuses. By defining atomic correlation, it infers atomic correlation violation to detect memory order misuse. Experimental results show that AtomVChecker successfully pinpoints 51 unknown memory ordering misuse. We hope our study will improve programmers' understanding of atomics and memory ordering issues, and help programmers build more secure applications.

X. Acknowledgments

This study is sponsored by the Fundamental Research Foundation for the Central Universities under Grant 2023RC29, Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing and Ant Group. Our sincere thanks go to the anonymous reviewers. Their insightful critiques have greatly improved this work. We are also grateful to the crate maintainers who responded constructively to our reports.

References

[1] The Rust Team. 2023. The Rust Programming Language. https://www.rust-lang.org/.
[2] Stack Overflow. 2017. Stack Overflow Developer Survey 2017. https://insights.stackoverflow.com/survey/2017/#most-loved-dreaded-and-wanted.
[3] Stack Overflow. 2018. Stack Overflow Developer Survey 2018. https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted.
[4] Stack Overflow. 2019. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019/#most-loved-dreaded-and-wanted.
[5] Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020/#most-loved-dreaded-and-wanted.
[6] Stack Overflow. 2021. Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021/#technology-most-loved-dreaded-and-wanted.
[7] Stack Overflow. 2022. Stack Overflow Developer Survey 2022. https://survey.stackoverflow.co/2022#technology-most-loved-dreaded-and-wanted.
[8] Stack Overflow. 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages.
[9] SnarkOs. 2023. A Decentralized Operating System for ZK Applications. https://github.com/AleoHQ/snarkOS.
[10] SnarkOs. influxdb. 2023. Scalable datastore for metrics, events, and real-time analytics. https://github.com/influxdata/influxdb.
[11] Sled. 2023. The champagne of beta embedded databases. https://github.com/spacejam/sled.
[12] TiKV. 2023. Distributed transactional key-value database. https://github.com/tikv/tikv.

[13] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding real-world concurrency bugs in go. In Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems(ASPLOS '19). Providence, RI, USA.

[14] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI '20). London, UK.

[15] Milind Chabbi and Murali Krishna Ramanathan. 2022. A study of real-world data races in Golang. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation(PLDI '22). San Diego, CA, USA.

[16] 2021. choosing between relaxed and acquire/release memory ordering in atomics. https://stackoverflow.com/questions/64973515/choosing-between-relaxed-and-acquire-release-memory-ordering-in-atomics.

[17] reddit. 2021. Find the right ordering for Atomics. https://www.reddit.com/r/rust/comments/p9a740/question_find_the_right_ordering_for_atomics/.

[18] Github. 2023. Will fetch_add with relaxed ordering cause update loss? https://github.com/m-ou-se/rust-atomics-and-locks/issues/2.

[19] reddit. 2020. Having a hard time understanding atomic operations in Rust. https://www.reddit.com/r/rust/comments/hskm11/having_a_hard_time_understanding_atomic/.

[20] RUSTSEC-2022-0006. 2022. {Iter, IterMut}::next used a weaker memory ordering when loading values than what was required. https://rustsec.org/advisories/RUSTSEC-2022-0006.html.

[21] RUSTSEC-2022-0029. 2022. Memory corruption due to weak memory ordering. https://rustsec.org/advisories/RUSTSEC-2022-0029.html.

[22] Performance Issues. 20223. Potential to weaken ordering for success case of compare_exhanges in race module. https://github.com/matklad/once_cell/issues/220.

[23] Mara Bos. 2022. Rust Atomics and Locks. O'Reilly Media, Inc.

[24] loom. 2023. Concurrency permutation testing tool for Rust. https://github.com/tokio-rs/loom.

[25] keh Akinyemi. 2023. Rust Concurrency Patterns for Parallel Programming.

[26] Steve Klabnik and Carol Nichols. 2023. The Rust programming language.

[27] Rust-nomicon. 2023. The Rustonomicon.

[28] Mara Bos. 2022. Comparing Rust's and C++'s Concurrency Library. https://blog.m-ou.se/rust-cpp-concurrency/.

[29] The C++ Team. 2023. Memory Ordering in C++. https://en.cppreference.com/w/cpp/atomic/memory_order.

[30] Hans Boehm and Paul E McKenney. P2055R0: A Relaxed Guide to memory_order_relaxed.

[31] Occlum. 2023. A library OS for Intel SGX. https://github.com/occlum/occlum.

[32] Servo. 2023. The memory-safe, parallel web rendering engine. https://github.com/servo/servo.

[33] Teaclave. 2023. An open source universal secure computing platform. https://github.com/apache/incubator-teaclave.

[34] Parking_lot. 2023. Compact and efficient synchronization primitives for Rust. https://github.com/Amanieu/parking_lot.

[35] Tokio. 2023. A runtime for writing reliable asynchronous applications with Rust. https://github.com/tokio-rs/tokio.

[36] Rayon. 2023. A data parallelism library for Rust. https://github.com/rayon-rs/rayon.

[37] Actix. 2023. Actor framework for Rust. https://github.com/actix/actix.

[38] Crossbeam. 2023. Tools for concurrent programming in Rust. https://github.com/crossbeam-rs/crossbeam.

[39] Once_cell. 2023. Rust library for single assignment cells and lazy statics without macros. https://github.com/matklad/once_cell.

[40] CVE. 2024. Common Vulnerabilities and Exposures. https://www.cve.org/.

[41] RustSec. 2024. Security advisory database for Rust crates. https://github.com/RustSec/advisory-db.

[42] Wikipedia. 2024. Data-flow analysis. https://en.wikipedia.org/wiki/Data-flow_analysis.

[43] Wikipedia. 2024. Dependence analysis. https://en.wikipedia.org/wiki/Dependence_analysis.

[44] Github. 2024. Relax SeqCst ordering in standard library. https://github.com/rust-lang/rust/pull/122729.

[45] Github. 2023. Change memory ordering in System wrapper example. https://github.com/rust-lang/rust/pull/106599.

[46] Github. 2021. Change Backtrace::enabled atomic from SeqCst to Relaxed. https://github.com/rust-lang/rust/pull/92139.

[47] Thread_local. 2023. Per-object thread-local storage for Rust. https://github.com/Amanieu/thread_local-rs.

[48] Hans-J Boehm and Sarita V Adve. 2008. Foundations of the C++ concurrency memory model. ACM SIGPLAN Notices 43, 6 (2008), 68–78.

[49] Bill Pugh. 2020. Reordering on an Alpha processor. https://www.cs.umd.edu/~pugh/java/memoryModel/AlphaReordering.html.

[50] Richard L Sites. 1993. Alpha AXP architecture. Commun. ACM 36, 2 (1993), 33–44.

[51] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. 2024. Understanding and Detecting Real-World Safety Issues in Rust. IEEE Transactions on Software Engineering (2024).

[52] Miri. 2023. An interpreter for Rust's mid-level intermediate representation. https://github.com/rust-lang/miri.

[53] loom. 2023. Concurrency permutation testing tool for Rust. https://github.com/tokio-rs/loom.

[54] Thomas Heartman. 2024. Understanding the Rust borrow checker. https://blog.logrocket.com/introducing-rust-borrow-checker/.

[55] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: detecting severe concurrency bugs through an effect-oriented approach. ACM Sigplan Notices 45, 3 (2010), 179–192.

[56] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2022. Detecting cross-language memory management issues in Rust. In European Symposium on Research in Computer Security(ESORICS '22). Copenhagen, Denmark.

[57] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: detecting atomicity violations via access interleaving invariants. ACM SIGOPS Operating Systems Review 40, 5 (2006), 37–48.

[58] MIRAI. 2023. Rust mid-level IR Abstract Interpreter. https://github.com/facebookexperimental/MIRAI.

[59] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles(SOSP '21). Virtual Event, Germany.

[60] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. ACM Transactions on Software Engineering and Methodology 32, 4 (2023), 1–21.

[61] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In Proceedings of the 2021 ACM SIGSAC conference on computer and communications security(CCS '21). Virtual Event, Republic of Korea.

[62] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 1–29.

[63] Weiyu Luo and Brian Demsky. 2021. C11Tester: a race detector for C/C++ atomics. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS '21). Virtual, USA.

[64] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI '19). Phoenix, AZ, USA.

[65] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 1–29.

[66] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. ACM SIGPLAN Notices 46, 1 (2011), 55–66.

[67] 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.