BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS USING FACEBOOK INFER FOCUSED ON ERRORS IN RCU-BASED SYNCHRONISATION

STATICKÁ ANALÝZA V NÁSTROJI FACEBOOK INFER ZAMĚŘENÁ NA CHYBY V RCU SYNCHRONIZACI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
DANIEL MAREK

SUPERVISOR
VEDOUCÍ PRÁCE
prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

25138

Student:	**Marek Daniel**
Programme:	Information Technology
Title:	**Static Analysis Using Facebook Infer Focused on Errors in RCU-Based Synchronisation**
Category:	Software analysis and testing

Assignment:

1. Get acquainted with principles of static analysis based on abstract interpretation. Pay a special attention to approaches proposed for finding problems in synchronisation of concurrent threads.
2. Familiarise yourself with the Facebook Infer framework, its support for abstract interpretation, and analysers for concurrent programs available currently in Infer.
3. Study principles of synchronisation of concurrent threads using RCU.
4. Design and implement an analyser in the Facebook Infer framework targeted at discovering synchronisation errors in programs using RCU.
5. Experimentally validate your analyser on suitably chosen non-trivial programs.
6. Summarise the achieved results and discuss possibilities of their further enhancements in the future.

Recommended literature:

- Rival, X., Yi, K.: Introduction to Static Analysis: An Abstract Interpretation Perspective. MIT Press, 2020.
- Blackshear, S., Gorogiannis, N., O'Hearn, P. W., Sergey, I.: RacerD: Compositional Static Race Detection. In: Proc. of OOPSLA'18, PACMPL 2(OOPSLA):144:1-144:28, 2018.
- Gorogiannis, N., O'Hearn, P.W., Sergey, I.: A True Positives Theorem for a Static Race Detector. In: Proc. of POPL'19, PACMPL 3(POPL):57:1-57:29, 2019.
- Harmim, D.: Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer. Master thesis, FIT, Brno University of Technology, 2021.
- Marcin, V.: Static Analysis Using Facebook Infer Focused on Deadlock Detection. Bachelor thesis, FIT, Brno University of Technology, 2019.
- McKenney, P.E., Fernandes, J., Boyd-Wickizer, S., Walpole, J.: RCU Usage In the Linux Kernel: Eighteen Years Later. ACM SIGOPS Oper. Syst. Rev. 54(1):47--63, 2020.
- McKenney, P.E., Walpole, J.: What is RCU, Fundamentally? Linux Weekly News, 2007. Available online: https://lwn.net/Articles/262464/. [checked on 29/9/2021]

Requirements for the first semester:

- The first three items and at least beginning of work on the fourth item.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Vojnar Tomáš, prof. Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 11, 2022
Approval date:	November 3, 2021

Abstract

Read Copy Update (RCU) is a synchronization mechanism, found mostly in the Linux kernel. Its sought-after features include almost zero overhead and high speed when reading shared memory. RCU has a set of rules of use, which need to be followed in order for the synchronization to work properly. To the best of our knowledge, there is no analyzer that sufficiently verifies if the RCU rules of use are adhered to. To overcome this, we propose a new analyzer, with the focus on finding violations in the use of RCU rules. The analyzer is based on static analysis and implemented as a module for the static analysis framework Facebook/Meta Infer. This platform was chosen because it provides scalability, which is needed when dealing with such an extensive software as the Linux kernel. The designed analyzer is capable of detecting multiple different ways in which the RCU usage rules can be broken, each causing either a race condition or a deadlock. It is also capable of generating warnings for situations when a deprecated function call is used or when the use of incompatible RCU reader and writer primitives is detected. The analyzer is the first of its kind. It may become a basis for future analyzer development in the field of Read Copy Update. Furthermore, it may be used as a test tool in the Linux kernel development cycle.

Abstrakt

Read Copy Update (RCU) je synchronizační mechanismus, který se primárně používa v jádře Linuxu. Mezi jeho vyhledávané vlastnosti patří téměř nulová režie a vysoká rychlost při čtení sdílené paměti. RCU má soubor pravidel používaní, která je potřeba dodržovat, aby synchronizace fungovala správně. Náš výzkum ukázal, že neexistuje žádný analyzátor, který by pořádně kontroloval dodržování pravidel používaní RCU. K překonání tohoto problému jsme navrhli nový analyzátor, který se zaměřuje na porušování pravidel používaní RCU. Analyzátor je založen na statické analýze a implementován jako modul pro nástroj pro statickou analýzu Facebook/Meta Infer. Tato platforma byla vybrána, protože poskytuje škálovatelnost, která je potřebná při práci s tak rozsáhlým softwarem, jakým je Linuxové jádro. Navržený analyzátor je schopen detekovat více porušení pravidel používaní RCU, z nichž každé vede buď na race condition, nebo uváznutí. Je také schopen generovat varování pro situace, kdy je použito volání zastaralé funkce nebo když jsou detekována nekompatibilní primitiva RCU čtecího a zapisovacího procesu. Analyzátor je první svého druhu a může se stát základem pro budoucí vývoj analyzátorů v oblasti Read Copy Update. Kromě toho může být použit jako testovací nástroj v cyklu vývoje jádra Linuxu.

Keywords

Read Copy Update, RCU, RCU rules violation, Facebook/Meta Infer RCU analyzer, Static RCU analyzer, RCU analyzer, RCU checker

Klíčová slova

Read Copy Update, RCU, Porušení RCU pravidel, Facebook/Meta Infer RCU analyzátor, Statický RCU analyzátor, RCU analyzátor

Reference

MAREK, Daniel. *Static Analysis Using Facebook Infer Focused on Errors in RCU-Based Synchronisation.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Read Copy Update (RCU) je synchronizační mechanismus, který se stal významnou součástí vývoje linuxového jádra. Své místo si našel především v kódu Driver, File System a Networking, ale je přítomen i v jiných částech linuxového jádra. RCU je vyhledáváno kvůli jeho minimální režii a vysoké rychlosti při čtení sdílené paměti. Je to možné díky tomu, že procesy nebo vlákna, které čtou sdílenou paměť, aniž by v ní prováděly jakékoli změny, mohou neustále běžet. To znamená, že je není potřeba zastavovat ani v situacích, kdy probíhá aktualizace sdílené paměti. Je to možné, protože aktualizace sdílené paměti se provádějí na kopii sdílené paměti, která bude publikována pro čtecí procesů nebo vlákna, jakmile bude připravena. Tento princip sebou přináší i dva problémy. Každá změna sdílené paměti může způsobit nekonzistenciu ve výsledcích, které daný produkt generuje, protože některé procesy anebo vlákna můžou stále vidět starou verzi vzdálené paměti a jiné můžou pracovat s tou novou. Taktéž je po každé změně vzdálené paměti potřebná synchronizace s ostatními procesy anebo vlákny předtím než může být stará vzdálená paměť uvolněná z paměti.

Samozřejmě existují určitá pravidla, jak se musí RCU používat, aby mechanismus fungoval tak, jak bylo zamýšleno. Lidé jsou však náchylní k chybám, které mohou způsobit porušení některých z těchto pravidel. Tato porušení mohou způsobit chyby synchronizace. Pokud taková situace nastane, je potřeba závadu najít a odstranit, což je často složitý a časově náročný proces. Čas je však cenným aktivem, a proto je nejlepší jej využít produktivně. Z tohoto důvodu je běžnou praxí používat nástroje, které urychlují a usnadňují proces hledání a odstraňování defektů.

Našli jsme šest unikátních způsobů porušení pravidel správného používání RCU, které jsme zobecnili do chybových vzorců. Čtyři z těchto vzoru jsme klasifikovali jako chyby, kde každá z nichž vede buď na race condition, nebo uváznutí. Další dvě vzory jsme klasifikovali jako varování, které je taktéž vhodné uživatelovi ukázat. Jsou to varování pro situace, kdy je použito volání zastaralé funkce nebo když jsou detekována nekompatibilní primitiva RCU čtecího a zapisovacího procesu.

Náš výzkum však ukázal, že existují velmi omezené možnosti, pokud jde o nalezení chyb na základe námi navrhnutých chybových vzorů. Až tak, že jsme nebyli schopni najít žádné řešení, které by bylo schopno odhalit porušení RCU pravidel na základě více než jednoho chybového vzoru. To znamená, že k pokrytí všech šesti z nich by bylo potřeba použít šest individuálních řešení, pokud by dokonce existovalo řešení pro každý ze šesti chybových vzorů.

K vyřešení nedostatku dobrých řešení navrhujeme statický analyzátor se zaměřením na správné použití RCU, který je schopen detekovat narušení na základě každého ze šesti chybových vzorů. Analyzátor je implementován pomocí rámce Infer.AI, který nabízí vysoce škálovatelnou analýzu, která je nezbytná při práci s tak rozsáhlým softwarem, jakým je linuxové jádro.

Testování analyzátoru mělo dvě fáze. Za prvé byl testován na ručně vyrobených příkladech pro ověření funkčnosti analyzátorů a za druhé byl testován na samotném linuxovém jádře. Analýza linuxového jádra se ukázala jako problematická, protože Infer není určen pro tento úkol. Podařilo se nám však zajistit, aby náš analyzátor fungoval na verzi 5-16.14 linuxového jádra. Bohužel jsme byli nuceni přeskočit část souborů, protože Infer je nedokázal zkompilovat. Přesto se nám podařilo analyzovat kolem 1300 souborů.

Analyzátor je teda první svého druhu a může se stát základem pro budoucí vývoj analyzátorů v oblasti Read Copy Update. Kromě toho může být použit jako testovací nástroj v cyklu vývoje jádra Linuxu.

Static Analysis Using Facebook Infer Focused on Errors in RCU-Based Synchronisation

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of prof. Ing. Tomáš Vojnar Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Daniel Marek
May 17, 2022

Acknowledgements

I would like to thank my supervisor prof. Ing. Tomáš Vojnar Ph.D. for his guidance. Further, I would like to thank Bc. Tomáš Beránek and Ing. Dominik Harmim for providing supplementary information and assistance.

Contents

Chapter 1

Introduction

Read Copy Update (RCU) is a synchronization mechanism that has become a significant part of the Linux kernel development. It found its place primarily in Driver, File System and Networking code, but it is present in other parts of the Linux kernel as well. RCU is sought after because of its minimal overhead and high speed when it comes to reading shared memory. It is possible thanks to the fact that processes or threads that read the shared memory without making any changes to it, are allowed to run constantly. It means that there is no need to stop them even in situations when the shared memory is being updated. It is possible because updates of the shared memory are performed on a copy of the shared memory, which will be published to the reading processes or threads once it is ready.

There are of course some rules on how the RCU has to be used, for the mechanism to work as intended. However, humans are prone to making mistakes which can cause some of these rules to be violated. These violations may cause defects to the synchronization, which most often result in a race condition and sometimes in a deadlock. If such situation happens, the defect needs to be found and removed, which is often a complex and time-consuming process. However, time is a valuable asset and therefore, it is best used productively. For this reason, it is a common practise to use tools that make the process of finding and removing defects faster and easier.

We found six unique ways of violating the rules of RCUs' proper use, which we generalized into error patterns. However, our research has shown that there are very limited options when it comes to finding violations corresponding to these patterns. To the point, where we were not able to find any solution that is able to detect violations based on more than a single error pattern. It means that to cover all six of them, six individual solutions would need to be used, if there even were solutions for each of the six error patterns.

To solve the lack of good solutions, we propose an static analyzer with focus on proper use of RCU that is capable of detecting violations based on each of the six error patterns. The analyzer is implemented using the Infer.AI framework that offers highly scalable analysis that is necessary when dealing with such an extensive software as the Linux kernel.

Chapter 2

Read Copy Update

Read Copy Update (RCU) is a synchronization mechanism, commonly found in the Linux kernel. It is often regarded as the successor of reader-writer locking because of their extensive similarity and the fact that RCU tends to outperform its counterpart. The comparison of these two mechanisms is discussed in Section 2.2.

RCU is compatible with both processes and threads, so the same principles can be applied in both of these cases. We will concentrate this thesis mainly on threads to avoid repetition. In RCU terminology, threads that only read from the shared memory are regarded as RCU readers and threads that also modify the shared memory are regarded as RCU writers or more commonly RCU updaters. There is one more type of threads in the RCU terminology they are regarded as RCU reclaimers and their purpose is to free the shared memory that is no longer accessible to any other threads. Reclamation of the shared memory is connected with nearly every update, and for this reason, it is common for a thread to start as an RCU updater and end as an RCU reclaimer. There are a few exceptions where no reclamation is needed, for example, an insertion of a new node to a list.

RCU uses a unique technique to synchronize its readers and updaters, which allows RCU readers to run in parallel with RCU updaters. This effectively means that RCU readers may work continually, and it is often regarded as the most significant advantage of RCU. The RCU synchronization is discussed in Section 2.1.2. RCU offers multiple flavors for both the Linux kernel and for the Userspace, each built for a specific need. RCU flavours are discussed in Section 2.5.

However, RCU is not always suitable its readers are the strong point of this synchronization mechanism, but its updaters and reclaimers not so much. They have to commit more resources to make continuous reads possible. Furthermore, RCU readers working in parallel with RCU updaters introduce inconsistency, as some RCU readers may still be working with no longer valid data. Types of applications suitable for the use of RCU are discussed in Section 2.4.

2.1 How does RCU work?

Read Copy Update introduces a unique technique of synchronization between its readers and updaters by adding new principles to both of them. The meaning of RCU readers and updaters is explained above. These principles need to deal with the fact that updaters work in parallel with readers.

If RCU updaters tried to change values in the shared memory directly, a race condition would be introduced. Read copy update as its name suggests, proposes a solution based on copying. So, when any RCU updater wants to update, for example, a certain data structure, it makes an exact copy first and leaves the current one intact. The RCU updater can make any changes to the copy without causing any problems as it is the only process with access to the copied data structure.

However, the problem has not been resolved just yet. There is a data structure with updated values, but RCU readers see the outdated one. RCU resolves this second problem just by swapping these two data structures. This swap is done by changing the value of a pointer, through which the RCU readers access the data structure. It is done atomically to eliminate any problems connected with situations where the value of the pointer is being read and changed at the same time.

At this point, it would seem that the update is done as the data structure has been updated and published to the RCU readers, but one important step remains. The now outdated data structure needs to be freed from the memory, but another problem is encountered. There is a chance that some RCU reader is still working with the outdated data structure, because RCU readers usually work with a locally saved pointer. They store the pointer locally because the one accessible to all the readers may be swapped anytime by any RCU updater.

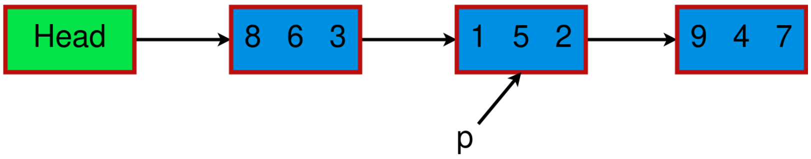
If an RCU updater freed the memory as soon as he published the new data structure, there is a chance that some RCU reader would try to access this already freed memory, which would cause problems. RCU avoids this situation by putting the RCU updater to sleep and waking it up when the updater is the only process with access to the outdated data structure. The explanation of how this process works can be found in Section 2.1.2. After waking up, the updater frees the memory and the update is complete.

These RCU principles are demonstrated on an example in Section 2.1.1, and a more in-depth explanation of how the RCU mechanism works can be found in the Linux kernel documentation [15].

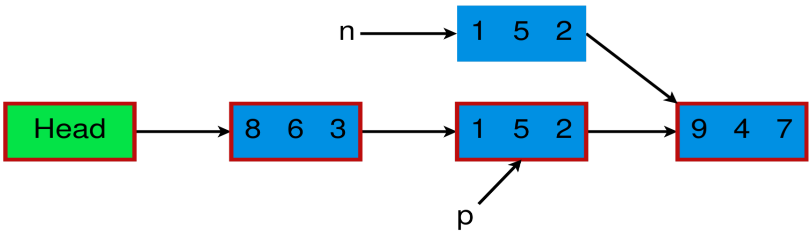
2.1.1 A Demonstration of the RCU Principles

We will demonstrate the main RCU principles using a series of figures. These figures represent an example of an update, namely an update of a node in a linked list. This combination was chosen, because an update of the RCU protected memory shows most of the key RCU principles and a linked list was chosen, because it clearly shows the implications of the RCU updates. The example is inspired by an article [32] written by one of the RCU founders Paul E. McKenney.

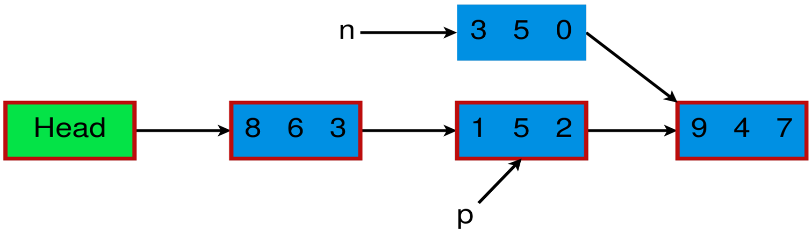
The figure below demonstrates the initial state. We start with a simple linked list, where we want to update a node pointed by the variable „p“. The red border symbolises that the node is accessible to at least one RCU reader.



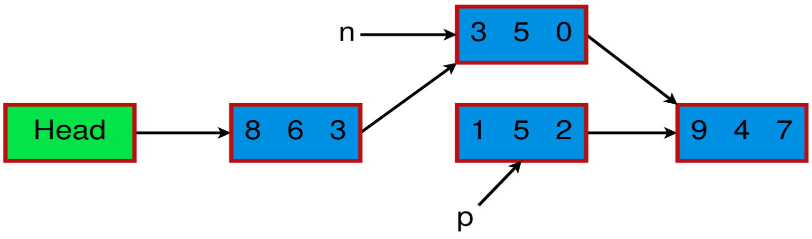
As discussed in Section 2, RCU readers may work continually, so an update of the current node would introduce a race condition. To avoid this race conditions, a copy of the old node is created for the RCU updater to work with. This situation is demonstrated in Figure below.



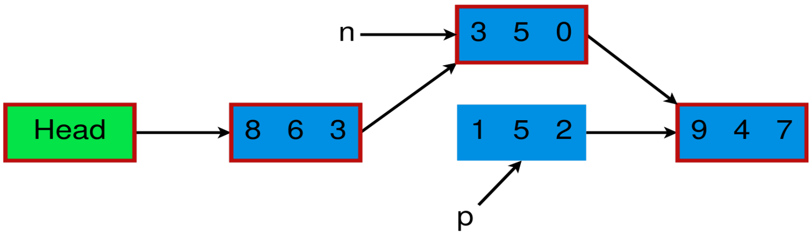
As this copy is only visible to the RCU updater, its values can be freely changed as shown in Figure below. After the update, the new node is ready to be published to the RCU readers.



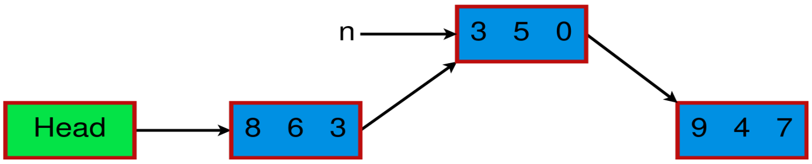
Using the „`rcu_assign_pointer(...)`" API call, the linked list is changed to contain the updated node. However, the memory that the old node uses, cannot be reclaimed yet. It may still be accessible to some of the RCU readers as demonstrated in Figure below. Another RCU API call is used, which puts the RCU updater to sleep and waits until the old node is no longer accessible to any RCU readers. It is worth noting that, at this point, RCU readers may see two different variants of the linked list. This fact introduces inconsistency to results produced by RCU readers, it is another sacrifice made to make the continuous reads possible.



The old node is no longer accessible to any RCU readers as shown in Figure below, so the RCU updater is awaken and it is free to reclaim the memory.



The memory is reclaimed and the RCU updater (or rather reclaimer) ends. There is once again only one version of the linked list as demonstrated in Figure below. As we can see, the inconsistency occurs only during a certain time after an update. It is one of the reasons why RCU is not recommended to be used in applications with frequent updates.



2.1.2 RCU Reader Synchronization

RCU reader synchronization is a process done by RCU updaters and it is needed in situations where an outdated element needs to be reclaimed. Reclamation of an element is a term from RCU terminology, it means that an element needs to be freed from memory, so the memory can be used again for other actions, usually other updates. A deletion and an update of an element are the situations where the RCU reader synchronization is needed.

The principles of the RCU update were explained in Section 2.1 and demonstrated in Section 2.1.1. We will concentrate on just the last part of the update, from the moment when the updated result is published to the readers until the reclamation of the outdated element happens. RCU divides this last part into three phases, as demonstrated in Figure 2.7.

The first phase is called „Removal". It is the part of the update that makes the element no longer accessible to any new RCU readers. For example, in a linked list, deletion of a node or a swap of a node for a new one would be considered as actions belonging to the „Removal" phase. This phase begins with the first action that makes an element inaccessible to the RCU readers and ends just before the reader synchronization starts.

The second phase is the heart of the RCU reader synchronization and it is called a „Grace Period". Its role is to make sure that the outdated element can be safely reclaimed. This section tends to be the longest time-wise, but it usually consists of a single RCU API call in RCU updaters. First action that this API call does is that it puts the RCU updater to sleep and the whole synchronization with the RCU readers is done by the RCU internally. If the situation does not allow for a process to be put to sleep, the situation is resolved through a callback function. In this scenario, the RCU updater ends by the RCU reader

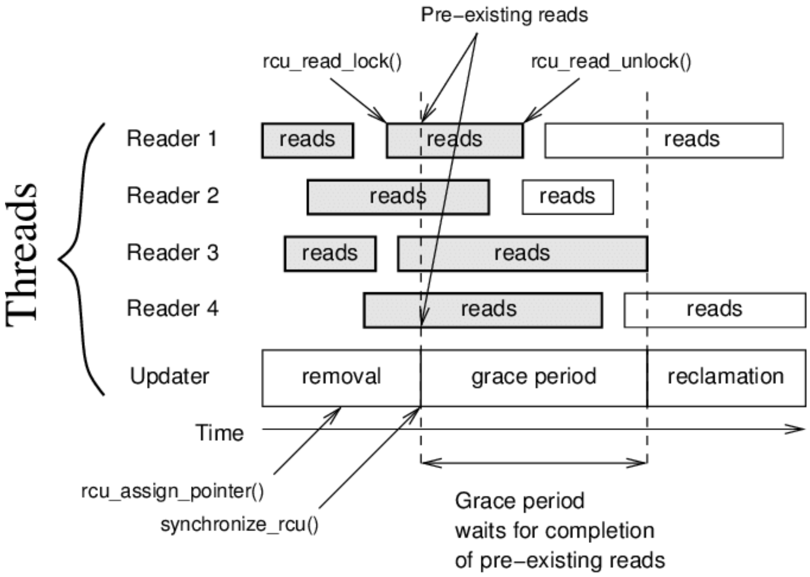


Figure 2.7: RCU reader synchronization is divided into 3 phases that represent steps needed for an outdated element to be safely freed from the memory. The „Removal“ section makes the outdated element inaccessible to all RCU readers that are not inside their critical section. However, there may be some readers whose critical sections started before the element was made inaccessible and thus may hold a reference to it. For this reason, the RCU updater, has to wait until all such RCU readers exit their critical section. The time spent waiting is regarded as a „Grace period“, and after it ends, the outdated element can be freed from the memory without causing any problems. The section responsible for cleaning the memory is regarded as the „Reclamation“ section. [Figure taken from [20].]

synchronization API call used and specifies which function should be called to reclaim the memory, when it is safe to do so.

The RCU reader synchronization is based on two principles. Firstly, RCU readers access the RCU protected memory exclusively through a local copy of the global RCU protected pointer, which they acquire inside their critical section. Secondly, they are allowed to access the shared memory through this local pointer only inside the critical section in which it was acquired. If all RCU readers adhere to these principles, it is safe to assume that only the RCU readers that entered their critical section before the Grace period started, may be holding a reference to the outdated memory. Therefore, the whole point of RCU reader synchronization is to wait, until all the RCU readers, that entered their critical section before the Grace period started, declare the end of their critical section. As a consequence the Grace period has no fixed length. This situation is demonstrated in Figure 2.7.

But an important question still remains, how does RCU determine that all the relevant RCU readers left their critical sections? There is a naive solution and then there is the real solution implemented in the Linux kernel. Lets start with the naive one. It relies on the fact that the classic RCU does not allow context switching during the RCU reader critical section and the fact that all RCU tasks need to run on a CPU. No context switching means that once the RCU reader critical section starts, the thread that started it holds the CPU until its critical section ends. All that the naive solution does, is that it tries to invoke a context switch on all of the existing CPUs by running an empty task on them. Once it succeeds, it can be sure that there is no active critical section that started before the RCU reader synchronization. The Linux kernel implementation is much more complex, because it needs to support all RCU flavors and interrupts, implement timers and be robust and secure enough to survive the Linux kernels environment. The explanation of how the Linux kernel implementation works is explained in a video[1] from the 2019 Kernel Recipes conference.

Last but not least, the last phase is called „Reclamation“ and consists purely of actions needed to reclaim all of the no longer accessible memory. It usually consists purely of one or more „free“ or „kfree“ statements, however, the procedure may be more complex in some cases.

2.2 Comparison with Reader-Writer Locking

RCU is most often compared to the reader-writer locking because both synchronization mechanisms try to achieve the same goals, but in a different fashion as demonstrated in Figure 2.8. Both mechanisms divide processes and threads into those that only read from the shared memory (Readers) and those that also make updates to the shared memory (Writers). Both allow readers to work in parallel with other readers, but they deal with updates differently.

In the case of reader-writer locking, the writer needs the shared memory just for itself, so it locks it. If any readers or writers want to work with the same memory, they have to wait until the current writer unlocks it. However, even if the writer holds the lock to this memory, it is not guarantied to be the only process with access to it. There may be some readers that started working with this memory before the updater managed to lock it. The fact that the writer needs the memory just for itself can create an awkward state where most of the readers and writers cannot work with the memory because it is locked. However, not even the writer holding the lock to this memory can work with it because it needs to wait for some readers, namely those that started reading before the writer expressed its interest into writing. To make the matters worse, the waiting is usually active, which means that it costs CPU time.

RCU treats updates differently, the main difference is that RCU updaters do not work with the shared memory directly. They first create a copy of the shared memory, for example, a copy of a node in a linked list, and this copy is only visible to the updater that created it. This allows changes to happen without stopping the readers or introducing a race condition. Once the memory is ready, it is atomically switched with the now outdated memory. This operation is performed by changing the value of a pointer, used by other processes to access this memory, to point to the new memory instead of the old one. The last step of the update involves freeing the outdated memory that involves synchronization with the reader and it is discussed in Section 2.1.2.

Both approaches have their own strengths and weaknesses. RCU managed to lower the overall overhead by putting all of it onto the updater/writer but not without a cost. RCU has a higher memory overhead and may produce inconsistent results, because some RCU readers may still be working with the outdated element. On the other hand, reader-writer locking suffers in performance as demonstrated in the article [25] by one of the RCU founders Paul E. McKenney.

[1] https://youtu.be/bsyXDAouI6E

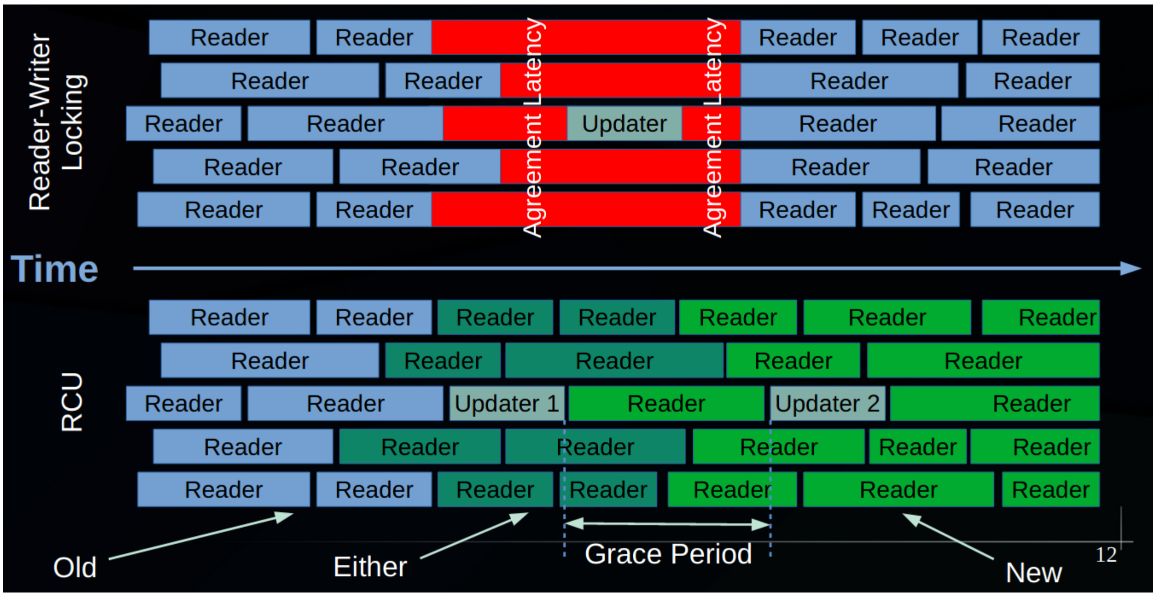


Figure 2.8: RCU and reader-writer locking have a different approach to updates. An updater in the Reader-Writer Locking mechanism needs an exclusive access to the soon to be updated memory, so it blocks other processes from using it. However, sometimes even the writer itself needs to wait because some readers started reading this shared memory before the writer expressed an interest into writing. All of this creates a significant overhead as there is only a single thread allowed to work with the shared memory during its update. An RCU updater on the other hand does not work with the element directly. All the changes are done to a copy of the element that switches places with the old element once it is prepared. It allows the RCU readers to work constantly because they always have an element to work with. There are however two problems. Some readers may be still working with the old element even after the new one becomes available, which means that they may produce outdated results and it also means that the updater cannot free the old element before it synchronizes with the readers. The synchronization represents an overhead, but it affects only the one updater. [Figure taken from [31].]

2.3 RCU Rules of Use in Code

In order for the RCU to work as intended, a set of rules of use has to be adhered to when writing an application. We will once again use the terminology of RCU readers, updaters and reclaimers, which is explained at the start of this chapter. These RCU rules of use in code will be explained on simple examples of RCU reader (Figure 2.9) and RCU updater (Figure 2.10) functions. These examples are inspired by the Linux kernel manual [15].

RCU Reader Rules of Use

An RCU reader follows just a few basic rules. In order to work with shared data, it needs to declare the start of a critical section. This prevents RCU reclaimers from reclaiming the memory immediately after an update. RCU readers should store a pointer to the shared memory locally as the value of the global pointer may change in the course of the critical section duration. The global pointer should be saved using the „`rcu_dereference(...)`"

API call, which ensures atomicity of this operation. It is needed because an RCU updater may try to change the pointers value at the same time as it is read.

Furthermore, it is important for all of the accesses to the shared memory to be performed inside the critical section because the shared memory may be reclaimed as soon as the critical section ends. However, any other operations should be performed outside the critical section to make the wait of the RCU reclaimer as short as possible. Last but not least, the critical section needs to be explicitly ended, for the same reason. An example of an RCU reader function is shown in Figure 2.9.

```
rcu_reader() {
    // definition of local variables
    rcu_data_type * p = NULL;
    ...
    // the beginning of the critical section
    rcu_read_lock();
    // safely store the value of the global RCU pointer
    p = rcu_dereference(g_ptr);
    // operations with the local pointer
    ...
    // the end of the critical section
    rcu_read_unlock();
    // other operations
    ...
}
```

Figure 2.9: A simple function demonstrating RCU reader principles.

RCU Updater Rules of Use

An RCU updater introduces just a few new rules. Basic RCU allows only a single update at one time, so synchronisation between multiple RCU updaters (if used) is needed. There is no RCU specific writer synchronization, so a spin lock is commonly chosen. The RCU updater needs to create a copy of an element that needs to be updated and work exclusively with it to not affect any RCU readers. Furthermore, after the new element is prepared, a pointer to it needs to be published to both the RCU readers and updaters and the old pointer needs to be made inaccessible to all processes. The „`rcu_assign_pointer(...)`" API call is used to perform both of these actions atomically.

The next action is to reclaim memory containing the old data. However, some readers may still have a pointer to this memory saved locally, so the RCU updater needs to wait until they end their critical section. For this reason the „`synchronize_rcu()`" API call is used. It puts the calling process to sleep and wakes it up after all critical sections that started before the call to „`synchronize_rcu()`" have ended. This operation may take a long time, therefore the RCU updater should unlock the spin lock before it starts, allowing other RCU updaters to work while it is sleeping. The process of synchronization with readers and subsequent reclamation of memory may be delegated to a different process if desired or needed. An example of an RCU updater function is shown in Figure 2.10.

```
rcu_updater() {
    // allocate new memory for a new element
    rcu_data_type * new = kmalloc(sizeof(rcu_data_type), GFP_KERNEL);
    // lock updates for other RCU updaters
    spin_lock(&update_mutex);
    // safely store the value of the current global pointer
    rcu_data_type * old = rcu_dereference(g_ptr);
    // initialization of the new element by copying the old one
    memcpy(new, old, sizeof(rcu_data_type));
    // update the new element
    ...
    // safely publish the new element
    rcu_assign_pointer(g_ptr, new);
    // unlock updates for other RCU updaters
    spin_unlock(&update_mutex);
    // synchronize with RCU readers
    synchronize_rcu();
    // free the old element
    kfree(old);
}
```

Figure 2.10: A simple function demonstrates the RCU updater principles.

2.4 When to Use RCU?

RCUs strengths revolve around its minimal reader-side overhead and its weaknesses focus mainly on the updates. Therefore, the viability of the RCU rises the more reads outmatch the updates when it comes to operations with the shared memory. Other important factors are consistency and freshness of results. Because of how updates in RCU work, it is common to have 2 or more versions of the shared memory at the same time, where different RCU Readers may see different versions of it. Therefore it is apparent that they will most likely not produce consistent results. The more the application needs fresh and consistent data, the more RCU struggles. To be fair, RCU provides support for fully fresh and consistent data, however it then suffers in performance. The Figure 2.11 displays the RCU viability using a graph.

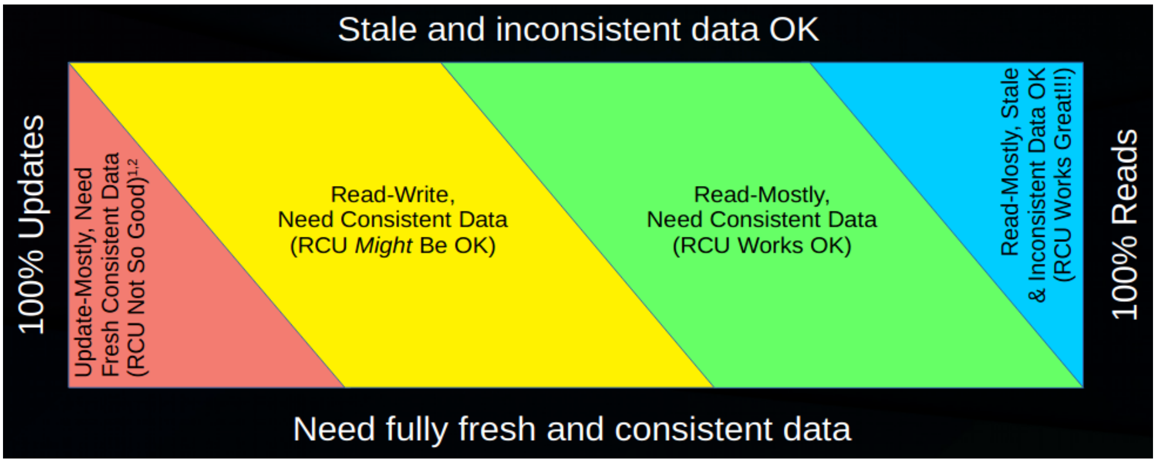

Stale and inconsistent data OK
100% Updates
Update-Mostly, Need Fresh Consistent Data (RCU Not So Good)1,2
Read-Write, Need Consistent Data (RCU Might Be OK)
Read-Mostly, Need Consistent Data (RCU Works OK)
Read-Mostly, Stale & Inconsistent Data OK (RCU Works Great!!!)
100% Reads
Need fully fresh and consistent data

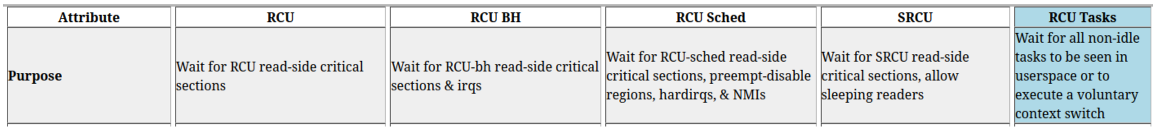
Figure 2.11: Viability of the Read Copy Update principle. [Figure taken from [30].]
.

2.5 RCU Flavors

RCU as a synchronization mechanism has been constantly evolving since its introduction into the Linux kernel back in the year 2002. As it started gaining popularity, it became apparent that just a single variant of RCU would not be enough to cover all of the developers needs. One of the biggest problems that was encountered revolved around the fact that the classic variant of RCU does not support sleeping or any kinds of interrupts inside the RCU reader critical section because it registers them as the end of the critical section. This limitation hindered a wider spread of RCU use, therefore the founders of RCU came with a solution. They implemented different variants of RCU according to the needs of the Linux kernel developers. These different variants of RCU are regarded as RCU flavors in its terminology. Moreover, despite the RCU was originally developed for use in the Linux kernel, there have been attempts to bring it to the Userspace as well, which led to an introduction of a new RCU variant. Each RCU variant offers multiple flavors to choose from and they will be introduced in the following sub-sections.

Linux kernel RCU flavors

According to the RCU API table [29] from the year 2019, there are currently 6 flavors in the Linux kernel version of RCU, which are displayed in Figure 2.12.



Attribute	RCU	RCU BH	RCU Sched	SRCU	RCU Tasks
Purpose	Wait for RCU read-side critical sections	Wait for RCU-bh read-side critical sections & irqs	Wait for RCU-sched read-side critical sections, preempt-disable regions, hardirqs, & NMIs	Wait for SRCU read-side critical sections, allow sleeping readers	Wait for all non-idle tasks to be seen in userspace or to execute a voluntary context switch

Figure 2.12: Various RCU flavors [Figure taken from [29].]

.

The first flavor is the classic RCU where sleeping and interrupts act as the end of the readers critical section, which may end the Grace period. If such situation occurs, the shared memory may be reclaimed sooner than it should be, and the interrupted reader may try to access an already freed memory once it resumes.

The RCU BH flavor adds support for hardware interrupts by not treating them as the end of the critical section, and the RCU Sched flavor extends this support to even more kinds of interrupts.

The SRCU flavor or Sleepable RCU is a unique flavor that allows sleeping inside the RCU reader critical section. This flavor requires the user to perform a few extra steps when writing an application. First of all, the user has to manually create and initialize at least a single `srcu` structure. This structure is used for synchronization between the RCU readers and updaters that need to specify which instance of the structure they are using. An updater would, for example, use the „`synchronize_srcu(&ss1)"` statement to synchronize using the `srcu` structure `ss1`. Secondly, when the RCU reader declares a start of the critical section, it receives an integer value that indicates during which Grace period the critical section started. This number is later used to determine for which Grace period was the critical section ended, and it is needed because the RCU critical sections can be nested. An RCU reader would, for example, use the „`int id = srcu_read_lock(&ss1)"` statement to declare the start of the critical section and the „srcu_read_unlock(&ss1, id)“

statement to declare the end of its critical section. A more detailed explanation of SRCU can be found in the article [26], which was taken as an inspiration for this summary.

The last RCU flavor is RCU Tasks that allows any tasks to run as long as they please, which can cause significantly long Grace periods, sometimes even in magnitudes of several minutes.

Userspace RCU Flavors

When it comes to Userspace RCU, threads that plan to use RCU need to explicitly declare it. For this reason, each such thread should start by using the „`urcu_<flavor>_register_thread()`" statement, which declares the point from which the Userspace RCU mechanism will keep track of all of its RCU-related statements. Furthermore, all of the threads using RCU should explicitly unregister from the RCU mechanism once they do no longer plan to use it. The Linux kernel variant infers these information automatically, and therefore eliminates problems where a thread using RCU does not synchronize with the others because the author forgot to register it.

According to the official website [21], the Userspace RCU offers 5 flavors: `memb, qsbr, mb, signal, bp`. The `mb` and `memb` RCU flavors both internally use memory barriers for synchronization on both updaters and readers sides and their primitives are used exactly the same way as the classic RCU Linux kernel variant. The official website [21] states that the use of memory barriers on both sides results in faster Grace period detection, but this approach loses some performance on the reader-side. The `memb` RCU is regarded as the more efficient and more flexible version of the `mb` flavor and it is the recommended choice when creating Userspace RCU applications. However, this flavor is only available if the kernel supports „`sys_membarrier()`". Otherwise, this flavor behaves exactly like the `mb` one if not defined differently.

The `qsbr` flavor handles reader-side synchronization the opposite way than all the other flavors. Qsbr does not declare the start and end of the critical section, but it rather declares whenever the reader enters a quiescent state. The quiescent state is a state that RCU readers enter whenever they leave their critical section, and it lasts until their next critical section begins. For this approach to work each RCU reader thread has to periodically declare that it entered a quiescent state to allow Grace periods to end. According to the official website [21], this approach offers the best performance for the reader-side, however, it introduces more intrusiveness to the code.

The `signal` flavor has the reader and writer synchronization internally based on sending signals instead of using memory barriers or declaration of a quiescent state, but it is used in the same way as both `memb` and `mb`flavors.

Finally, the `bp` flavor stands for the bulletproof flavor, and it was designed in order to make hooking to application easier for a tracing library by allowing it to hook itself onto an application without a need to change the code. This approach introduces more overhead, and therefore suffers in performance on both reader and updater sides.

Chapter 3

Our Road to an RCU Analyzer

No product just magically appears, it starts with an idea and requires a lot of research, determination and hard work before it sees the light of the day. Our analyzer is no exception, our road started with a curiosity about a synchronization mechanism called Read Copy Update. It was a topic that nobody in our surrounding knew anything about, and therefore it became a main focus for our research. Our most important findings about RCU were discussed in Chapter 2.

The next step was to identify what kind of an analyzer would be beneficial for RCU developers. To achieve this, the website [24] maintained by one of the RCU founders Paul E. McKenney was scouted. This website serves as a crossroad to most of the RCU related work that is available online. We discovered that, when it comes to verification and validation, the articles concentrate on RCU itself, mainly on checking if RCU functions as expected. However, means for testing/analyzing the way RCU is used are rather limited as discussed in Section 3.1. So, we decided to develop such an analyzer, but there were still several steps needed before the development of the analyzer itself could start.

Firstly, there was a need to familiarize ourselves with even these rather limited solutions. This step was important to make sure that we would not just create a replica of an already existing product and also to obtain a product to compare our solution with. Our findings are discussed in Section 3.1.

Secondly, since we decided to create a static analyzer, there was a need to research static analysis and its techniques, so we could choose one that fits our needs. Static analysis is discussed in Section 3.2, and our chosen static analysis technique Abstract Interpretation is discussed in Section 3.2.1.

The next important step was to decide which Static Analysis framework to choose for the development. We decided upon Facebook (now Meta) Infer because it is based on Abstract Interpretation and offers high scalability, which is needed when dealing with such a huge software as the Linux kernel. More about Infer is discussed in section 3.3.

The last step was to determine error patterns and design for our analyzer. The design and error patterns are introduced in Chapter 4.

3.1 Available Solutions

As RCU is used primarily in the Linux kernel, we decided to focus on solutions that offer support for the Linux kernel testing and analysis. According to the Kernel Testing Guide [10], there are several options available.

Firstly, the Linux kernel provides the kselftest and KUnit frameworks, which are used to create tests that focus on isolated parts of the Linux kernel. The focus on isolated parts can be seen as both the pro and con according to the situation. On one hand, it helps to make tests more efficient as they can be optimized for a certain situation. On the other hand, when one decides to apply it for testing the use of RCU, it means that every time RCU is used in a different situation or different environment, new tests need to be written. Furthermore, these frameworks do not directly offer support for RCU, so if a developer wants to track the RCU state, he or she would need to use another library or tool to achieve this goal.

Secondly, there are testing tools with various functionality separated into Code Coverage and Dynamic Analysis tools, according to the Kernel Testing Guide [10]. Code Coverage tools are not relevant for our use case, so we will concentrate exclusively on the second group. The guide [10] provides us with a long list of Dynamic Analysis tools, out of which we decided to pick KCSAN, KFENCE and lockdep as they are the most relevant.

KCSAN (The Kernel Concurrency Sanitizer) [8] is a tool that focuses on finding data races and KFENCE (Kernel Electric-Fence) [9] provides a low-overhead detector of memory issues, which is able to detect use-after-free errors among other memory errors. Data races and use-after-free errors are common consequences of RCU rules of use violations, and these dynamic analyzers provide a way of detecting them. Furthermore, these analyzers are able to perform the analysis efficiently because they were developed and optimized for use in the Linux kernel. However, neither of these analyzers is able to determine which RCU rules of use violations caused these defects. The whole process of finding the root cause is on the developers' shoulders. This may be a significantly time consuming activity, especially in situations where the problem was caused by an RCU rules of use violation, some time before the defect was encountered.

Lockdep [13] is a locking correctness validator that can detect deadlocks and other locking-related defects. RCU itself is immune to lock-based deadlocks, but lockdep can be used to find other types of defects caused by RCU rules-of-use violations. Lockdep knows the current RCU state because it tracks which tasks enter and leave their RCU reader critical section and it keeps this state separately for each RCU Linux kernel flavor used in the code, according to its Linux kernel documentation [12]. Furthermore, lockdep keeps track of other types of locks too, so it knows about critical sections on the RCU updater side as well. There are even multiple RCU API calls capable of working with the lockdeps RCU state. An overview of them can be found in the RCU API table in the article [29] in the section „Validation". For example, the „rcu_read_lock_held()" statement returns a positive boolean value if it determines that it is called within an RCU reader critical section. This bool value can be used for other RCU statements that check if the given condition is fulfilled to decide whether they can securely perform their tasks.

Additionally, if the Linux kernel is built with the „CONFIG_PROVE_RCU" flag, lockdep will dynamically check whether conditions for use of the RCU dereference primitives are met before executing them. If, for example, the „rcu_dereference()" statement is used, lockdep checks whether at least one condition for the safe dereference of a global pointer is met. The dereference is safe if the accessed memory cannot be reclaimed while accessing it, which is guaranteed if it is done inside a correct RCU reader critical section. There are of course other methods of stopping the shared memory from being reclaimed, such as holding a writer lock for the corresponding shared memory. If the lockdep determines that it is not safe to access the memory, it creates a Lockdep-RCU Splat, which holds information that is useful for debugging, such as stack backtrace and overview of locks that were held by the process or thread at that time. An example of a Lockdep-RCU splat can be found on its documentation website [11]. A more in-depth explanation of lockdeps support for RCU can be found in the Linux kernel documentation [12] that served as an inspiration for this overview.

An advantage of the lockdep dynamic analyzer is that its support for RCU was added by the founders of this synchronization principle. They expanded the lockdep with the RCU reader critical section detection and developed new RCU API calls that would make use of the information gathered by the lockdep. The steps leading to the lockdep adaption by the RCU founders can be found and explained in the article [27] by Paul E. McKenney. The support started in the year 2010, and the idea was to use this tool to detect the missing starts of RCU reader critical sections by finding situations where the „rcu_dereference" statements were used without any protection. The RCU API has had 2 major updates since that year and there have been a few changes to the lockdep RCU API calls, but the idea and functionality of the lockdep remains the same. The fact that lockdep is still maintained by the RCU developers in addition to it being designed and optimized for use in the Linux kernel make it an excellent candidate for use.

On the other hand, keeping the information about locks and the RCU state by the lockdep introduces overhead and this overhead hinders performance of the product, especially if the „CONFIG_PROVE_RCU" flag is set. In this case there is an added overhead to every „rcu_dereference()", which is a very common operation for products using RCU. Furthermore, the tool is able to detect just a single type of RCU rules of use violations, which greatly limits its usefulness. Moreover, the lockdep tool is a dynamic analysis tool, so it suffers the shortcomings of the dynamic analysis itself, such as the fact that the dynamic analysis can miss problematic situations because the paths leading to them may not be executed during the analysis. Lastly, when lockdep detects a defect, it creates a Lockdep-RCU splat that can be used to find the error, but the finding of the root cause is once again on the developers' shoulders.

The Development tools for the kernel [7] section of the Linux kernel manual expands the Kernel Testing Guide [10] with three more tools, namely Checkpatch, Coccinelle, and Sparse. Checkpath is a Perl script, capable of detecting trivial style violations, one of them being the use of RCU deprecated API calls in the code. This feature can be useful in some cases, however, it covers just a tiny portion of RCU related defects, and therefore Checkpath cannot sufficiently cover the developers' needs.

Coccinelle [6] is a tool for pattern matching and text transformation, it has many uses in the Linux kernel, but none related to RCU.

Sparse [14] is a static semantic checker for C programs and it is used mostly for lock and type checking. According to the article [27], Sparse offers support for a single type of RCU defects. It can detect situations where the global RCU pointer is accessed without the use of the RCU dereference primitive. This is a valuable information because it covers one of the RCU rules of use. However, using Sparse for this task is inconvenient in that the user has to manually tag all the relevant RCU pointers with the „__rcu" tag in order for Sparse to take them into account. It requires more than a basic understanding of RCU to determine which pointers are relevant, and it also requires to make direct changes to the code, which may be significantly time-consuming or forbidden. Sparse uses static analysis, which means that it does not suffer the same problems as the tools before, but, of course, static analysis has its own shortcomings too. It is more inclined to finding false positives[1], which can lead

[1] Incorrectly identified defects.

to waste of time in situations where the developers try to find problems that do not exist. Our analyzer is also based on static analysis. However we cannot compare our solution with that of `Sparse` because they both specialize in different kind of problems. `Sparse` concentrates solely on the RCU pointers, while our analyzer detects problems connected with improper use of RCU function calls.

There is one more way of testing RCU infrastructure in the Linux kernel. There are a few kernel modules that are available under the `„RCU Debugging"` category in the Linux kernel Configuration. There is the `RCU_SCALE_TEST` kernel module that runs performance tests, there is the `RCU_TORTURE_TEST` kernel module that runs torture tests[2], and there is the `RCU_REF_SCALE_TEST` kernel module that runs performance tests on various read-side synchronization mechanism and compares their results with RCU. These kernel modules test RCU but in a different way than we need.

To summarize, there are multiple ways of testing and analyzing applications using RCU in the Linux kernel, but only two of them are related to the RCU rules of use namely, `lockdep & Sparse`. Both of them offer support for the same RCU rules of use violation, namely, „Unprotected RCU dereference" of global RCU pointers, but each detects these violations in different circumstances. From this research we came to a conclusion that an RCU analyzer that would be able to cover most of the RCU rules of use 'would be the most useful.

3.2 Static Program Analysis

According to [33] publication, static program analysis is the art of reasoning that focuses on the behavior of computer programs and it is able to do so without actually running them. It has found its place in multiple products such as compilers and standalone static program analysis tools. In compilers, it is used for both syntactic and semantic analysis as well as for the optimization of the final machine code. When it comes to the standalone static program analysis tools, static analysis can be used for detection of many kinds of defects, like unreachable lines of code, uninitialized variables, potential data races, deadlocks, and many more.

The important factors when it comes to static program analysis are soundness and completeness as well as the speed of the analysis. Starting with the speed, if the time needed for an analysis in a static program analysis tool is too high, nearly nobody will want to use it. One of the reasons being that it will not be fast enough to fit into the real-world software development cycle as stated in [33].

Secondly, the completeness of an analysis is the factor that shows what percentage of the found defects corresponds to the real defects in the code. In other words, it answers the question: „If a defect was found, how likely was it detected correctly ?" If the completeness is too low, a lot of errors may be identified incorrectly, which can lead to developers wasting their time trying to fix problems that do not actually exist. As a result, they may not want to use the product in the future.

Lastly, soundness of an analysis states what percentage of defects in the code will be found by the analysis. In other words, it answers the question: „If there is a defect in the code, how likely will it be found ?" Once again if the soundness is too low, the analysis may miss a significant portion of defects, and therefore lose attractiveness for the developers.

[2] Tests that make the product run at or near full capacity for an extended length of time. Definition inspired by [34].

Ideally, every static program analysis should achieve perfect soundness, completeness with little time needed for the analysis, but it is unreachable in the real life and the tools using static program analysis need to achieve some balance of these factors. If the soundness and completeness are a priority, very detailed analysis needs to be performed. This involves keeping highly detailed information about the source code and its state as well as deploying multiple heuristics to reduce the number of incorrectly identified defects. The problem is that the detailed information can create a significant memory overhead, and the use of multiple heuristics may require a significant processing time. This may create a situation where the analysis is able to produce highly accurate results for smaller-size products but fails to produce any results for bigger and more complex applications where the memory and processing time requirements become unacceptable or even unsatisfiable.

Therefore, it is common for the analysis to keep less detailed information and carefully select which heuristics will be used, but this approach creates another problem. For some defects, it may be impossible to determine if they are real or if they are just consequences of the analysis inaccuracies. If the analysis tends more towards higher soundness, it will report these types of defects even if there is a risk that they may be incorrectly identified. On the other hand, the analysis with focus on completeness would report exclusively the defects that are highly probable to be confirmed and ignore the rest. It is common for real life static program analysis tools to prioritize one of this factors at the cost of the other.

The analysis inaccuracies happen with the more detailed analysis too, but in smaller amount. The inaccuracies happen because it is not always possible to determine the state of the source code accurately, especially if it relies on data that is not available before the code is executed. There are of course other factors that contribute to the static program analysis success, such as computational complexity, scalability or memory overhead, but we will not discuss them in this thesis.

According to [33], static program analysis may be based on various different principles, but we are using the abstract interpretation because the static analysis framework Infer.AI that we decided to use for the implementation is based on this principle.

3.2.1 Abstract Interpretation

Our overview of abstract interpretation is inspired by the article [19]. A more formal and detailed explanation can be found, e.q., in [17, 18, 16]. We first illustrate principles of abstract interpretation on a series of figures. Starting with a demonstration of concrete semantics of a program in Figure 3.1, followed by a demonstration of abstract interpretation of these concrete semantics in Figure 3.2 to better demonstrate how the abstract interpretation works. The last two figures demonstrate two problematic situations where the Abstract state is too broad, shown in Figure 3.3(a) and not broad enough, shown in Figure 3.3(b). Subsequently, we then present abstract interpretation and its ingredients more formally.

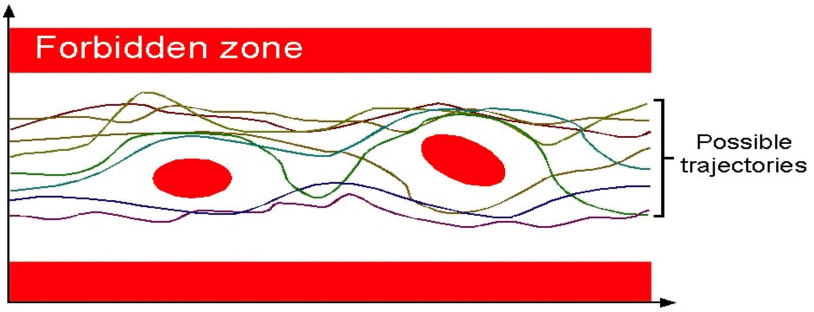


Figure 3.1: This figure represents the concrete semantics of programs. The lines demonstrate trajectories of all possible program executions for all environments where the program can be executed. There may be zones that introduce a safety or other vulnerabilities if any program execution trajectory crosses them. These zones are demonstrated with the red color in Figure. [Figure taken from [19].]

.

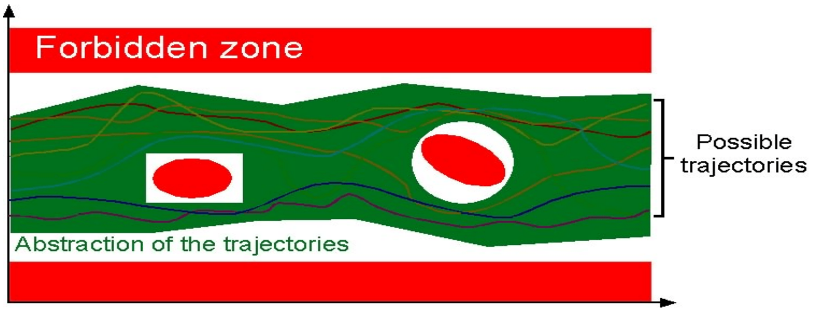
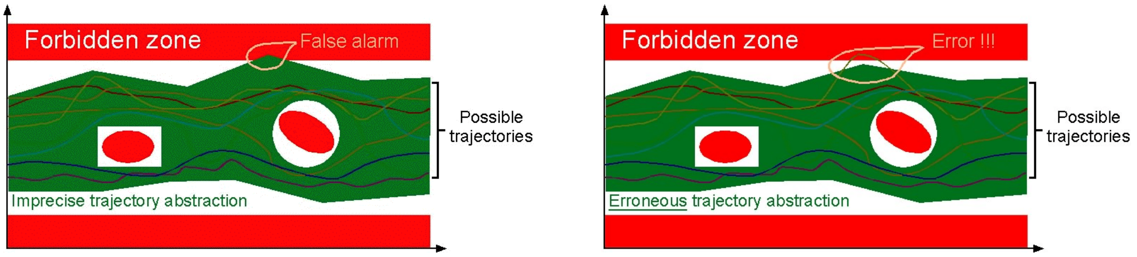


Figure 3.2: This figure illustrates the impact of an abstraction of all the possible trajectories. It is apparent that this abstraction includes some trajectories that cannot happen during concrete program execution. This is caused by the fact that abstract interpretation usually over-approximates the concrete program state. [Figure taken from [19].]

.



(a) Too broad

(b) Not broad enough

Figure 3.3: These figures represent two possible problems of abstract state approximation. In the first case(a), the analysis would find a non-existing vulnerability, and in the other case(b), a real vulnerability would be missed. It is possible for both of the problems to appear at the same time because the approximation may be too broad for some parts of the program and not broad enough for others. The challenge of abstract interpretation is finding the right approximation for the given situation. [Figures taken from [19].]

Ingredients of the Abstract Interpretation

There are multiple Ingredients of the Abstract interpretation, according to the [22, 23]. Our overview is taken from [23] with small changes.

- *Abstract domain*
  - a set of *abstract contexts*,
  - an abstract context represents a set of program states (typically used to represent a set of program states reachable at some program location).
- *Abstract transformers*
  - for each program operation there is a corresponding transformer that represents the effect of the operation performed on an abstract context.
- *Join operator*
  - combines abstract contexts from several branches into a single one.
- *Widening*
  - performed on a sequence of abstract contexts appearing at a given location to accelerate obtaining a *fixpoint*[3]
- *Narrowing*
  - may be used to refine the result of *widening*.

Abstract Interpretation formally

Abstract interpretation I of a program P with the instruction set `Instr` is a tuple

$$I = (Q, \sqcup, \sqsubseteq, \top, \bot, \tau)$$

where

- Q is the *abstract domain* (domain of *abstract contexts*),
- $\sqcup : Q \times Q \to Q$ is the *join operator* for accumulation of abstract contexts,
- $\sqsubseteq \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \iff x \sqcup y = y$ where
  - $(Q, \sqsubseteq)$ is a *complete (join semi-)lattice*,
- $\top \in Q$ is the supremum of $(Q, \sqsubseteq)$,
- $\bot \in Q$ is the (single) infimum of $(Q, \sqsubseteq)$,
- τ : `Instr` $\times\, Q \to Q$ defines the *abstract transformers* for particular instructions required to be *monotone* on Q for each instruction from `Instr`.

The *soundness* of abstract interpretation may be guaranteed using *Galois connections*. [This subsection is taken from [23] with small changes.]

[3] A fixpoint of a function $f : A \to A$ is an element $a \in A$ if and only if $f(a) = a$. [Taken from [22].]

Galois Connections

Galois connection is a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ such that:

- $\mathcal{P} = \langle P, \leq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are *partially ordered sets* (posets),
- $\alpha : P \to Q$ and $\gamma : Q \to P$ are functions such that $\forall p \in P$ and $\forall q \in Q$:

$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q$$

[This subsection is taken from [23] with small changes.]

3.3 Facebook/Meta Infer

The official Infer website [1] classifies Infer as a static program analyzer, which supports Java, C, C++ and Objective-C. It is written in the OCaml programming language and it offers two types of analysis frameworks. There is a separation logic bi-abductive analysis framework, which is primarily used to detect problems caused by null pointer dereferences and resource and memory leaks. The second analysis framework is based on abstract interpretation and it is primarily used for finding a wide range of problems (over/under-overflow, uninitialized variables, dead code, concurrency-related issues, performance-related issues, etc.). RCU rules of use violations fall into the concurrency category, and therefore we decided to use the second one.

Abstract Interpretation Framework

According to [2], the Infer.AI (Infer Abstract Interpretation) is a framework for quickly developing analyzers based on abstract interpretation. It provides all the necessary architecture for the analysis and the developer needs to define only an abstract domain and abstract transformers in addition to join, widen and less or equal operators. The less or equal operator is used to determine the order of abstract states in the abstract domain. If for example, the abstract state is represented by a set, the less or equal operator determines if one set is a subset of the second one. A set is a subset of another if all of its elements are contained in the other set.

The analyzers can be developed to be intraprocedural or interprocedural. The intraprocedural analyzers are only capable of finding defects contained in a single procedure as these types of analyzers do not save information about the procedures that have already been analysed. If, for example, a lock was locked in one procedure, but unlocked in a different one that was called from it, these analyzers would report a defect for both of these procedures although there is of course no defect as one is allowed to work with locks across multiple procedures.

On the other hand, interprocedural analyzers create a summary of the procedure once it is analysed. A big advantage of the Infer.AI is that it stores this summary in the results database and this procedure does not need to be analyzed again because the caller needs just its summary. Thanks to this, the first analysis of a program is the most time and resource demanding because all procedures need to be analyzed in order to get their summary. But after, the cost of the analysis is only determined by the number of procedures that need to be reanalysed, which can be a much smaller number. There are just two reasons why a procedure needs to be reanalysed, either it has been directly changed or it is a direct or indirect caller of a changed procedure. The change in a procedure may cause its summary to change and because the caller uses it to determine its abstract state, its summary may change as well.

A summary is a pair consisting of an abstract pre-state and an abstract post-state. The pre-state specifies under which conditions the function may be called while the post-condition then describes the result. However, in some cases, the pre-state does not need to be used. If an interprocedural analyzer is faced with the same situation as above, where a lock is locked in one procedure and unlocked in a procedure called from it, this type of analyzer may be designed such that it will handle the situation correctly. It is possible thanks to the fact that Infer.AI analyses procedures according to the call graph from the bottom up. This means that when the Infer.AI analyzes the procedure where a lock is locked and encounters a procedure call, it analyzes this called procedure first before continuing the analysis of the caller. So if it encounters a call to a procedure that unlocks the lock, it analyzes this procedure and its summary can be used in the caller to determine that the lock was correctly unlocked before the procedure that did the locking ended. The bottom up approach is shown and explained in Figure 3.4 on a call graph.

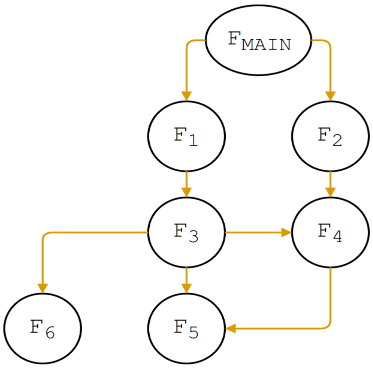


Figure 3.4: This figure represents a call graph where the arrows point from the caller to the called procedure. Infer.AI implements a bottom-up approach for the analysis, which means that the procedures that are not callers themselves are analysed first, followed by their callers and they are followed by their callers and so on. If there are multiple procedures called from a single caller, the order of their analysis is determined by their call order, meaning that the procedure that is called first will be analysed first as well. Furthermore, if a procedure has more than one caller, it is only analyzed once, because after it already has a summary that can be used. In case of an recursive call, Infer.AI analyzes only the caller and ignores the recursion. If it did not, Infer.AI would analyze the function until the whole analysis crashed because there would be nothing stopping it from diving into the same function over and over again. [Figure taken from [22].]

.

Analyzers Available in Facebook Infer

The Facebook Infers documentation [1] has a page for each type of analysis that is available for use. There is around twenty-six official analyzers, but some of them are no longer maintained, some are experimental and new ones keep appearing. The focus of the analyzers spreads across multiple areas, such as out-of-bounds array accesses, memory safety, cost, functions side effect detection and more. The whole list of available analyzers along with their description, types of detectable defects and examples can be found in the Infers documentation [1] on the Infers' official website. Some of these analyzers are more relevant for our research, namely Pulse, RacerD and Starvation.

Starting with `Pulse`, its documentation page [3] states that it is used for memory and lifetime analysis. It is able to detect defects like use-after-free, constant-address-dereference, null-pointer-dereference, memory leaks, and more. The only relevant type of defect detected by `Pulse` is use-after-free because it can be caused by a violation of RCU rules of use on either the RCU reader or updater side. However, `Pulse` is not designed for analysis of concurrent applications which means that it would most likely produce inaccurate results.

Secondly, the `RacerD` documentation page [4] classifies it as an analyzer for thread safety. It also states that it is used to identify data races, which are the most common consequences of violating RCU rules of use. The problem is that this analyzer does not support analysis for code written in the C language, according to its documentation [4]. As RCU is used primarily in the Linux kernel, where the C language code is prevalent, it means that `RacerD` cannot be used for our needs. Moreover, the analysis it implements is heavily tuned for programs that synchronize by classical locks.

Finally, `Starvation` is, according to its documentation page [5], used to detect situations where no progress is being done because of concurrency errors. Deadlock and starvation are typical errors detected by this analyzer. `Starvation` as an analyzer supports concurrency, but none of its detectable defect are connected with violations of RCU rules of use, because RCU does not use typical locks for its reader-side synchronization. A deadlock can happen but in an RCU-specific way, which is not supported by this analyzer. However, even if these situation were relevant for RCU, Starvation once again does not support its defect detection for code written in C language.

Chapter 4

RCU-rules-of-use Analyzer original design

We now proceed to the original design of our analyzer. We start by describing various error patterns related to violations of RCU rules of use that we identified and that our analyzer will look after in Section 4.1. Secondly, we describe how we apply abstract interpretation to check for identified error patterns. This is done by describing our design for various ingredients of abstract interpretation in Section 4.2. Lastly, we describe the design of the top-level logic in Section 4.3. This top-level logic describes the algorithms and principles used to determine defects and eliminate false positives.

4.1 Error Patterns

The RCU rules of use that are discussed in Section 2.3 determine the proper way this synchronization mechanism should be used when writing code. We found multiple ways in which they can be violated and we decided to generalize them into error patterns that will be used by our analyzer to detect violations of the correct way of using RCU. There are four error patterns that we decided to classify and report as errors and two that we classified as warnings because they represent situations where RCU is used in a not advised manner. We decided that it is better to report these situations as well because they may become problematic in the future.

Incomplete RCU Reader Critical Section

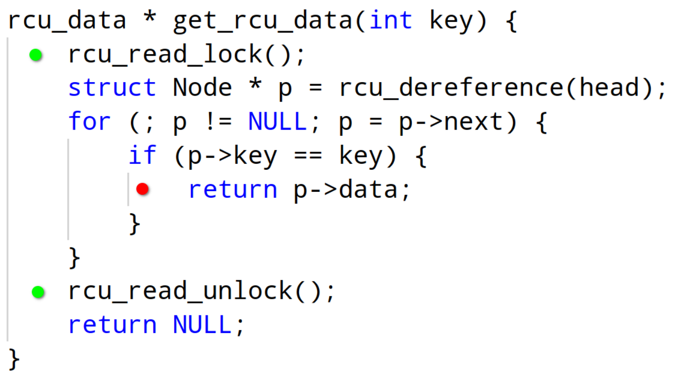
There are three types of problems when it comes to RCU reader critical sections completeness. Either there is a situation where the critical section is not properly started or it is not properly ended or possibly both. This type of problem happens mostly in functions that alter the RCU reader critical section and have more than a single exit point or where the start or the end of the critical section is hidden behind a condition, or when these functions do not behave consistently when it comes to the critical section start and end.

When a function has more exit points, it becomes more likely that one can miss the declaration of the end of the critical section. This is true especially in situations when a function is complex, long and when some of these exit points were added in the later development, possibly by a different developer than the one who created the original function.

When it comes to the declaration of the critical section start or end being hidden behind a condition, it is fair to say that it brings more trouble than worth. It can easily happen that the conditions used in the code do not do not work as expected because there is a program state that the developer did not account for. Moreover even if the conditions work as expected in the current version of the program, it may not be the case in the later ones.

Finally, when functions that alter the RCU reader critical section do not alter the critical section consistently, it becomes increasingly harder for developers to keep track of when the critical section needs to be started and ended. Some functions may require to be called within the callers critical section, some may declare their own critical section, some may declare just the start of the critical section and expect another function to declare the end or the other way around. There is also a situation where the called function declares the end of the critical section only if it returns a negative result, which forces the caller to hide the end of the critical section behind a condition.

There are different problems associated with missing declaration of the start and the end of an RCU reader critical section. If the critical section is not properly ended, it prevents the RCU reclaimer from being awaken after a call to „synchronize_rcu()". If the „call_rcu(...)" statement is used instead, the callback function cannot be invoked, meaning that there will be no reclamation of no longer used memory. Most of the RCU flavors use a timer that invokes a function that forcefully ends RCU reader-side synchronization and allows the memory reclamation once the synchronization duration exceeds a set upper bound, but this solution should be used only as a fallback and fixing the missing end of the critical section should be a priority. A function that both misses the end of critical section and contains an inconsistent critical section behavior is shown and explained in Figure 4.1.



```
rcu_data * get_rcu_data(int key) {
  ● rcu_read_lock();
    struct Node * p = rcu_dereference(head);
    for (; p != NULL; p = p->next) {
        if (p->key == key) {
          ●  return p->data;
        }
    }
  ● rcu_read_unlock();
    return NULL;
}
```

Figure 4.1: This figure represents a function that handles an RCU reader critical section differently when it finished successfully and when it does not. The developer either forgot to add the declaration of the end of critical section to the red-marked exit point or more likely thought that the caller will want to work with the returned data. While this presumption may be correct, it does not make the design of this function justified. The caller has most likely already its own critical section started and it may want to call this function multiple times, if it wants to get data from multiple nodes of the linked list that is used to store RCU data in this program. If this is the case, the caller is forced to declare the end of the critical section after each successful call to this function. If the developer does not notice this fact, the caller may start multiple critical sections without even knowing it. A much better design for this function would be to either not work with the critical section at all or declare its end at all exit points from the function.

.

On the other hand, If the critical section is not properly started and its end is declared anyway, it causes two types of problems. It results in an unprotected RCU dereference that is discussed in Section 4.1, and the declaration of the end can disrupt a different critical section. RCU reader critical sections can be nested, so the additional declaration of the end may cause the callers critical section to end prematurely, leading to other problems. There is also a situation where no critical section is active but its end is declared. This leads to a sort of weird situation where it ends the next critical section as soon as it starts, which is a sort of an unexpected behavior. Thankfully, RCU reports these situations to the user with a warning message every time the application tries to end a non-existing critical section. However, that happens only at run-time and it can happen that such a situation arises only rarely and so the situation would be unnoticed for a long time.

Unprotected RCU Dereference

As discussed in Section 4.1, this problem happens when the start of the RCU reader critical section is missing. However, this is not the only way this problem can happen as the RCU dereference is also used on the RCU updater side. This error pattern covers all the situations where RCU protected memory is accessed without any protection, which usually means that it is accessed outside of a critical section. This type of problem usually happens either because of problems associated with the critical section or when the developer is not aware that the RCU dereference needs to be protected. The developer may know that the dereferences in RCU readers need to be protected from the RCU reclaimers but he or she can miss the fact that the dereferences in the RCU updaters need to be protected as well. Figure 4.2 presents an unprotected RCU dereference on the RCU updater side and explains why it is problematic.

When an Unprotected RCU dereference happens, it introduces a race condition between an RCU reclaimer and the process or thread where the unprotected dereference happened. As is discussed in Section 2.3, it is only safe to access the RCU protected shared memory inside a critical section or when there is a guarantee that the accessed shared memory cannot be reclaimed while the process or thread still uses it.

If the RCU reclaimer „wins“ the race, meaning that the reclamation happens before the access to the shared memory, the thread or process accessing the shared memory will try to access an already freed memory, which can be highly problematic. The behavior of accessing already freed memory is undefined, meaning that different machines and systems may behave differently.

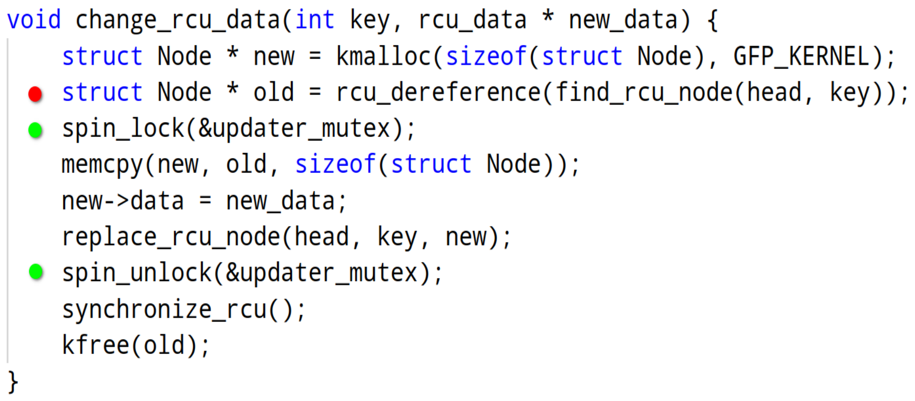
In an ideal case, this access will not result in a crash. Reads in this case will result in obtaining nonsensical values, while writes will not overwrite anything important.

In a less ideal case, the system forbids this access and the program crashes with a segmentation fault because the process or thread tried to access a restricted area of the memory.

In the worst case, the product does not crash immediately but only after there has been some damage done. Either the nonsensical read values cause an undefined state, that the program cannot handle or the write to the freed memory overwrites something important.

Missing RCU Reader Synchronization

As is explained in Section 2.1 and shown on an example in Section 2.1.1, RCU needs to execute reader-side synchronization before it reclaims the outdated memory. If this synchronization is missing, there is no protection for RCU readers, who are still work-



```
void change_rcu_data(int key, rcu_data * new_data) {
    struct Node * new = kmalloc(sizeof(struct Node), GFP_KERNEL);
  ● struct Node * old = rcu_dereference(find_rcu_node(head, key));
  ● spin_lock(&updater_mutex);
    memcpy(new, old, sizeof(struct Node));
    new->data = new_data;
    replace_rcu_node(head, key, new);
  ● spin_unlock(&updater_mutex);
    synchronize_rcu();
    kfree(old);
}
```

Figure 4.2: This figure shows a dangerous situation where the „rcu_dereference()“ statement is called before the critical section starts. It introduces a possibility for two or more RCU updaters to hold a pointer to the same memory. This situation happens every time when two updaters want to find the same RCU node and the updater that was called second manages to obtain the pointer to this RCU node with a call to „rcu_dereference()“ before the first one manages to update the pointer by publishing a new element with a call to „replace_rcu_node(...)“. If two or more updaters managed to obtain a pointer to the same shared memory, there are two ways it can become problematic, depending on when the first updater manages to free the memory. If it manages to free the memory before the second one enters the critical section, the second updater will try to copy from an already freed memory. This may lead to either the new node having nonsensical values that will be published or to a crash caused by a segmentation fault. On the other hand, if the second updater manages to copy the data before it is freed, it can happen that the new node has some outdated data as it is not copying the newest version. However, both of these situations result in a problem because both will try to free the same memory at the end. This behavior is undefined, but in most cases, it results in a crash.

.

ing with the outdated element. The synchronization is performed with a call to either „`synchronize_rcu()`" or „`call_rcu(...)`" or similar statements for other RCU flavors.

This type of mistake happens as an error from the side of a developer. Either he or she is not aware that this synchronization is needed for RCU or it is simply forgotten. Missing RCU reader synchronization results in unprotected RCU dereference, which is discussed in Section 4.1 with all of its consequences.

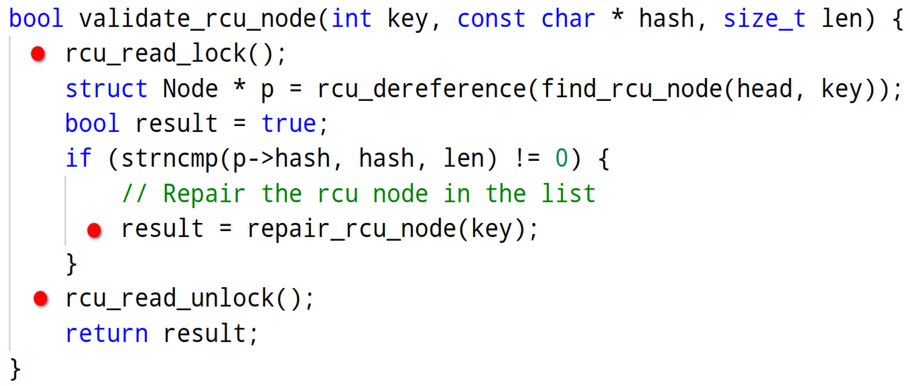
RCU deadlock

RCU introduces a special kind of deadlock. As discussed in Section 2.1.2, the call to „`synchronize_rcu()`" puts the caller to sleep, and it is awaken when all the critical sections that started before this call declare an end of their critical section. However, what happens if the synchronization itself is invoked within a critical section?

It depends on the RCU flavor. In some flavors, this causes a deadlock of the RCU reclaimer process or thread because the synchronization waits for the end of the critical section from which it is called, but this critical section will end only if the synchronization finishes. It is a mutual exclusion because both actions need the other action to finish first.

For other flavors, the call to a synchronization statement behaves as a context switch that causes an end of the critical section within which it is called. This situation, on the other hand, creates problems connected with premature termination of the critical section, leading most commonly to an unprotected RCU dereference, which is discussed in Section 4.1, or to an additional declaration of the end of the critical section, which is discussed in Section 4.1.

Synchronization inside an RCU reader critical section happens mostly in situations when an RCU updater function is called by an RCU reader. The developer may not realize that there is a reader-side synchronization hidden within the updater function and therefore he or she may call this function from the readers critical section. An example of this type of defect is shown and explained in Figure 4.3.



```
bool validate_rcu_node(int key, const char * hash, size_t len) {
  ● rcu_read_lock();
    struct Node * p = rcu_dereference(find_rcu_node(head, key));
    bool result = true;
    if (strncmp(p->hash, hash, len) != 0) {
        // Repair the rcu node in the list
      ● result = repair_rcu_node(key);
    }
  ● rcu_read_unlock();
    return result;
}
```

Figure 4.3: This figure presents a problematic situation where a function that makes changes to the RCU protected memory is called within a reader critical section. We can assume that the „repair_rcu_node(...)“ function tries to replace an invalid node with a new valid one. However, as is known, the invalid node cannot be freed from the memory without synchronizing with the RCU readers first. So, if the „repair_rcu_node(...)“ manages to repair a node, synchronization is invoked within a critical section. In this case, it would not cause a deadlock because the classic RCU flavor is used. Instead, the synchronization would act as the end of the critical section because it represents a context switch that the classic RCU flavors does not follow. In this particular example, no unprotected dereference would be caused, however, the declaration of the end of the critical section at the end of the function would cause problems explained in Section 4.1.

Use of Deprecated RCU API Calls

One of the warnings that we decided to show to the user represents a situation where an RCU API call is encountered that is marked as deprecated or removed in the latest RCU API table. This situation happens mostly when code was written a significant time ago and there have been just a few or no updates since. This warning can be useful in a situation where the application wants to start using a new version of RCU to check whether the update does not cause any problems in this manner.

Mismatch of RCU Flavours

Finally, yet another warning that we decided to display represents a situation where two or more RCU flavors are used at the same time. It usually means that the flavour of RCU used to declare the reader critical section is not compatible with the flavor of the primitive used for reader-side synchronization. This type of problem commonly happens with updates that decide to switch from the used RCU flavor to a new one because it suits their needs better. The problem may be caused a lack of understanding of RCU and its flavors or simply because some of the RCU statements were missed during the switch to a different version.

This exact situations happened in the Linux kernel development in the year 2018, where after switching from the classic RCU to the RCU-Sched flavor, the synchronization stopped working because only the reader-side primitives were changed. An email from Linus Torvalds asking for help with this situation can be found in both [28] and [30] presentations. As a consequence, changes were made to make the use of RCU easier in order to prevent the mismatch of RCU flavors.

We decided to classify this defect just as a warning, because it is not forbidden to use more than one RCU flavor at the same time, it is just not advised.

4.2 Abstract Interpretation Ingredients

We now proceed to our proposal of using abstract interpretation to detect the above described problems. In particular Section 4.2.1 presents the abstract context, Section 4.2.2 presents abstract transformers and Section 4.2.3 presents abstract operators.

4.2.1 Abstract Context

The first component that we needed to design is the abstract context where the design was based upon on a few assumptions. We assumed that we need to hold 3 types of information.

Firstly, we need to keep track of the RCU state during a functions analysis. Further we also need to save the final RCU state reached in the analysed function for other functions to use. We realized that a single part of the abstract context would be enough for both, because during the analysis of the given function it can hold the current state, and after the analysis is finished, it automatically holds the final RCU state in the given function. In our design this part of the abstract context is called „Postconditions“ or shortly „Post“. It is used for two purposes: first it is used by the abstract transformers to determine violations of RCU rules of use during the analysis of the given function and second it is used to check what the analyzed function left behind after it ended. For example, it may hold information that a function ended with a still active RCU reader critical section. This may be a defect or an intended result, it is up to the caller to decide. Without this information, it would be impossible to keep track of RCU reader critical sections that begin in one function and end in another.

Secondly, there needs to be a part of the abstract context that keeps information about what needs to happen before the function is called. If for example, the analyzed function declares only the end of the critical section it is safe to assume that it needs to be called within an active critical section. Similarly, if the function contains a call to „`rcu_dereference()`" without declaring the start of the critical section, it is once again safe to assume that the critical section needs be active before this function is called. In our design this part of the abstract context is called „Preconditions“ or shortly „Prec“.

Lastly, there needs to be a part of the abstract context that holds information about the problems that were found in the analyzed function. In out design this part of the abstract context is called „Problems“. There was also a need to determine how these 3 parts of the abstract context will be saved, and we decided to create a structure containing them. So the *Abstract Context* is a structure containing three parts and it looks like this:

- *Postconditions*: the RCU state during and at the end of the function,
- *Preconditions*: the RCU state needed to call this function,
- *Problems*: problems detected within the function.

Preconditions and Postconditions

There was also a need to determine how the parts of the abstract contexts will look like. Both the pre and post conditions need to store the RCU state, and since there are multiple RCU flavors, we needed to use a data structure that can hold multiple RCU states. We were deciding between a hash table and a set, where the set was chosen because it allows each element to be stored exactly once, which is beneficial, because we want to keep exactly one RCU state for each RCU flavor that is used inside the analyzed function or across multiple functions.

The RCU state itself is composed of two parts namely, lock name to determine which lock is used and lock score that tells us the locks state meaning how many times it is locked/unlocked. For example, lock score equal to two tells us that the lock is locked twice and it also needs to be unlocked twice in order for all the critical sections to end. The *RCU state* structure hast two parts and looks like this:

- *Lock name*: lock identification,
- *Lock score*: state of the lock.

Problems

When it comes to problems, multiple of them can be detected within the analyzed function. For this reason, a set was used because we want to keep each problem exactly once to avoid reporting of the same problem multiple times. Each problem is composed of a few parts that hold the necessary information.

Firstly, a problem needs to keep its description that will be reported to the user. Secondly, it needs to hold information about where the defect happened. We decided to save the name of the function and the line of code where the problem was detected. Lastly, we needed to save the error pattern under which the problem falls to report more useful information to the user. The *Problem* is a structure that hold four parts and looks like this:

- *Description*: description of the problem for the user,
- *Function name*: the name of the function where the problem was found,
- *LoC*: line of code where the problem was found,
- *Error pattern*: the corresponding error patter.

4.2.2 Abstract Transformers

Next, we needed to design the abstract transformers. We started by determining which concrete statements are relevant for our analysis and we came with two main categories.

Critical Section RCU statements

The first big category consists of RCU statements that start and end the RCU reader critical section. This concerns RCU read locks and unlocks for every single RCU flavor including the user-space ones. The transformers for this category are as follows:

- *RCU read lock*: increment the corresponding lock score,
- *RCU read unlock*: decrement the corresponding lock score; if it reaches a negative value, create a precondition and a problem.

We decided to create a problem for the situations where the lock score reaches negative values because it introduces undesirable behavior. However, if this function is called by a caller with an active critical section, the score is not actually negative. For this reason, we decided to create a precondition that once met allows the detected problem to be removed.

Other RCU Statements

The second big category consists of other relevant RCU statements. RCU dereferences, reader synchronization primitives, and the deprecated or removed RCU statements are all relevant and the category once again incorporates these statements from all the RCU flavors. The transformers for this category are as follows:

- *RCU dereference*: check whether it is called within a corresponding critical section; if it is not, create a precondition and a problem.
- *RCU reader synchronization*: check whether it is invoked within a corresponding critical section; if it is, create a problem.
- *Deprecated and removed RCU statements*: create a problem.

If an RCU dereference is called without a protection of the critical section, the situation can still be resolved if the critical section was started by the caller. For this reason, we create both a problem and a precondition. If the precondition is resolved, the problem is removed as well.

On the other hand, when an RCU reader synchronization is invoked within an active critical section, it is strictly a problem. Theoretically, it could be resolved if the caller „pre-unlocked“ the critical section by using an additional RCU read unlock, but this type of behavior is not wanted. For this reason, we create just a problem to report.

4.2.3 Operators

The abstract interpretation framework of Infer supports three types of operators: join, widen, and less or equal. It does not support the narrowing operator, so there is no need to design it.

Join operator

We needed to decide what to do when different states come from different branches for all three parts of an abstract context. The *join operation* behaves differently for the different parts of abstract context, we discuss below why it behaves as we now describe:

- *Post*: create a new set, where lock score is picked for each flavor from one of the sets according to the following condition:
  - if both scores are positive, take the higher value,
  - if both scores are negative, take the lower value,
  - if one score is equal to zero and the other is not, take the non-zero value,
  - if one score is positive and the other is negative, create an error and take the value from either set.
- *Prec*: create a new set containing all preconditions from both sets.
- *Problems*: create a new set containing all problems from both sets.

When it comes to postconditions, we decided to always take the state that is more problematic because we do not want to miss a potential error. For example, if one set declares that the RCU critical section was started twice and the other declares that it started only once, the twice started critical section is more problematic, so it will be taken as the final joined state.

A situation where one set holds a negative score and the other holds a positive one is highly unusual and rather problematic because both scores lead to different kinds of problems. We decided to create an error to inform the developer of this strange behavior and take either of these values.

When it comes to problems and preconditions, the two sets can have some of the same items, but thanks to the fact that it is a set, there will be no duplicates in the final combined result.

Widening operator

Widening operator is used when a loop or recursion is encountered to shorten or enforce the final state computation. We, however, do not expect the RCU operations to be repeated in such volumes that it would cause a problem. Therefore, our widening operator calls the join operator instead. This way the end result is much more accurate than if it was over-approximated.

Less or equal operator

This operator has a simple design. It compares all three parts of abstract contexts for both abstract contexts that it was given to compare. If it finds that all the elements of one abstract contexts are already contained in the second abstract contexts, it declares the first abstract contexts as a lesser one or as a equal one if both contexts contain the same elements. However, if there is even a single element that is unique for both of these abstract contexts, these two sets are taken as not comparable and therefore neither is lesser.

4.3 Analysis Logic

The analysis logic can be divided into three parts. There is the logic of the analysis as a whole, there is a logic that dictates how the abstract contexts reached at the end of analyzed functions will be turned into a summary, and lastly there is a logic that focuses on what the caller takes and uses from the summary of a called function for its own abstract context computation.

Overall Logic

The overall logic builds upon the Infer.AI core principles, mainly the fact that a program is analyzed according to the call graph from bottom up. This practically means that if a call to a function $f2$ is encountered during analysis of function $f1$, Infer will dive into the called function $f2$ and analyze it before continuing with the analysis of the caller $f1$. Infer can and often does dive several levels deep because during the analysis of the called function that it dived into, another call to a function is encountered and so on. The main advantage of this approach is that, during the analysis of a function we already know how the called functions alter the RCU state.

According to these principles, we assumed that all that is needed is to start the analysis for a single function that serves as an entry point, and all the functions called from it will be analyzed too because they have to be called directly or indirectly from this entry point. And we assumed that the entry function will be obtained from Infer. After the analysis of the entry function ends, we will have all the unresolved problems to report, because the resolved problems will not be propagated higher as they will be removed by the callers. Furthermore we assumed that the functions that are unused do not need to be analysed. The reason being that an analysis of such functions could be particularly inaccurate. For instance, If we analyzed a wrapper function, which is used to start the critical section without its caller we would get a false error, saying that the critical section was not properly ended. The steps of the *overall logic* are as follows:

1. *Start the analysis* on the entry point function with an empty abstract context, the analysis performs these tasks:
   - Let Infer.AI *dive into the called functions* and *analyze them.*
   - *Create summaries* of the called functions after their analysis finishes.
   - *Use summaries* of the called functions in the callers.
2. *Finish the analysis* of the entry point function.
3. *Join abstract contexts* from all entry functions exit points.
4. *Check* if there are any *active critical sections*; if there are, create a problem for each of them.
5. *Report all problems* to the user.

Summary Computation

The summary computation consist just of a single operation. It checks the current RCU state, and if it finds that the RCU reader critical section is still active, it creates a problem. All other parts of the abstract context stay the same, because they are used and altered during the analysis.

Applying Summary

Callers apply summaries of all the functions that were called from them in the order in which they were called. This needs to be done to determine how the called functions affected the abstract state. The steps of *applying the summary* are as follows: (we discuss them a bit more below)

1. Check which preconditions were met and *remove problems* resolved this way.
2. *Add the unresolved problems and preconditions* into the callers abstract state.
3. *Update the callers abstract state* according to the callers postconditions.

Applying a summary starts by removing problems that have been resolved, because they should not be propagated to the caller. This action is done by checking the preconditions against the callers current RCU state. If, for example, the precondition stated that the caller needs to have an active critical section and it is really the case, this precondition can be removed and all the corresponding problems with it.

The next step consists of propagating the unresolved problems and preconditions to the caller. The unresolved problems need to be propagated because they need to get to the entry function to be reported at the end of the analysis. The unresolved preconditions need to be propagated too because some of the problems may still be resolved by another caller.

Lastly, the caller needs to adjust its RCU state according to the final state of the called function. For example, the function may cause the end of the callers critical state, and if the analysis did not account for this change, it might either miss out on some errors or detect non-existing ones.

Chapter 5

Implementation

This section describes the implementation of the RCU-rules-of-use Analyzer which is based on the original design that was presented in the previous chapters. It focuses on the problems that we faced during the implementation and how we managed to solve them. As a result, the final analyzer somewhat differs from the original design. So, the implemented version can be taken as the latest design of our analyzer. The reasons for these changes are one of the points discussed in this chapter.

Furthermore, this chapter contains a section focusing on the parts of the abstract context (Section 5.1) and the Analysis Logic (Section 5.2). These sections have the same titles as in the previous chapter that focuses on the design of this analyzer. This is done to make it easier to draw parallels between the original design and the latest implemented design for these parts.

5.1 Abstract Interpretation Ingredients

The abstract interpretation ingredients are once again divided into three parts. The implemented abstract context is discussed in Section 5.1.1, the abstract transformers in Section 5.1.2, and the abstract interpretation operators are discussed in Section 5.1.3.

5.1.1 Abstract Context

We start with how the *implemented abstract context*, it is a structure containing four parts and it looks like this:

- *Problems*: problems detected within the function.
- *Postconditions*: RCU state during and at the end of the function.
- *Function Calls*: all function calls within the function.
- *Process Description*: Infer.AIs CFG node of the function.

The problems and postconditions are the only parts that are in both the original design and in the final implementation of the Abstract Context. Their importance is discussed in Section 4.2.1.

The abstract context original design also contained a member called preconditions that was later removed. We realized that it will be better if the problems themselves contained their cause, which can be used to determine if the problem has been resolved. Removing the pre-conditions has even made removing problems less complicated because before, once a precondition was met, there was a need to determine which problems were associated with this precondition. Now, all that is needed is to check for each problem if its cause is still valid and if it is, propagate this problem to the caller.

There are two new parts of the abstract context. The first part is called „Function Calls“ or shortly „FunCalls“, and, as its name suggests, it holds all function calls that were encountered inside the analyzed function. This part is used to determine which functions are „top-level“, meaning which functions do not have a caller. The need for this new part will be explained in Section 5.2. The second new part of abstract context is called „Process Description“ or shortly „ProcDesc“, and it is the Control Flow Graph node representing the analyzed function. It holds information about the analyzed function, such as its name, starting line of its code, parameters, return type and more. It is used during defect reporting to supply the information that the Infer.AI needs.

Postconditions

Postconditions need to hold the RCU state during the analysis. We decided to keep this part as a set as discussed in Section 4.2.1. The set holds all encountered *RCU states*, which consist of the following parts:

- *Lock name*: lock identification.
- *Maximal score*: maximal state of the lock.
- *Minimal score*: minimal state of the lock.
- *Access Path*: Infer.AIs specific identification that states how the lock was accessed.
- *Line of Code*: location of the lock in the source code.

The lock name is used to distinguish between different types of locks, sometimes between multiple RCU locks and nearly always between the reader- and updater-side locks.

There has been a notable diversion from the original design when it comes to the lock score part of the RCU state. It got divided into two parts that represent an interval of possible lock scores. We decided for this change because it represents the lock score more accurately.

There are two new parts that both identify the lock in the code. The access path is an Infer.AI specific identifier of the code elements. For a lock, the access path represents the way this lock was accessed in the code. For example, an access path may be $X.Y$ which means that the element Y was access through the element X. The element X is usually a structure that holds a lock. It is kept because Infer.AI requires it for reporting. The second new part is the line of Code where the lock was first encountered. It may hold the Line of Code of the start or the end of the critical section according to what is encountered first.

Problems

Problems are also represented as a Set. Where each of its items is a structure containing seven parts that are as follows:

- *Is Lock Problem*: determines whether the problem is connected with locking.
- *Is Lock Needed*: determines whether the problem is a missing lock.

- *Problematic Lock*: a problematic lock with its score (if the problem is connected with locking).
- *Process Name*: the name of the function where the problem was encountered.
- *Problems Line of Code*: the location of the problem in the code.
- *Problem Name*: description of the problem to report to the user.
- *Issue*: Infer.AI specific problem classification.

Problems now hold significantly more parts than it was the case in the original design. It still contains all of the original parts, the „Function name“ from the original design is now called „Process Name“, the „LoC“ is „Problems Line of Code“, the „Description“ is „Problem Name“ and „Error pattern“ is „Issue“. There are however three new parts, which are here because the cause of the problem was moved from pre-conditon to the problem itself.

The first part is called „Is Lock Problem“, and it tells us whether the problem is related to locking. If it is not, the analysis knows that it the problem can represent either the „missing RCU reader synchronization“ or the „deprecated or removed RCU API call“ error only. Therefore, it does not need to check the problematic lock.

The second part is called „Is Lock Needed“, and it tells us whether the cause of the problem is a missing start of the critical section. If it is the case, it has to represent the „Unprotected RCU dereference“ error and once again the analyzer does not need to check the problematic lock.

The third part is called „Problematic Lock“, and it stores the problematic RCU state if there is any. In the case of an unended critical section, it would hold a RCU state containing a lock with either both minimal and maximal scores or just the maximal score higher or equal to one. If the previous two parts do not determine the error type this part will be checked to see if the problem was not resolved.

These three new parts will be likely reworked in the future development because the current solution is rather overly complicated. The „Issue“ part is capable of determining whether the problematic lock needs to be checked. It is therefore likely that only the „Problematic Lock“ part will stay.

Function Calls and Process Description

The Function Calls are represented as a set, where each item consists purely of a functions name. The process description is the CFG node of the analyzed function, which is implemented internally in the Infer.AI and not by us, so we will not dive further into it.

5.1.2 Abstract Transformers

The abstract transformers underwent some changes that expand their behavior. There is also another category added to the original two.

Critical Section Statements

Transformers for the RCU reader critical statements were expanded with other non-RCU locking statements. So, the new transformers support the typical locking statements for both the Linux kernel and the user-space along with the RCU specific reader-side locking. It was done because the original design would detect non-existing „Unprotected RCU dereference“ errors on the updater-side. The problem was that the original design did not account for other than RCU critical sections. And since RCU updaters use RCU dereference but not the RCU critical sections, their dereferences would be falsely classified as unprotected. The expanded version is as follows:

- *Any lock*: increment the corresponding lock score,
- *Any unlock*: decrement the corresponding lock score.

It is worth noting that we also do no longer immediately create a problem if, after an unlock, the lock score reaches a negative value immediately because these types of problems are now created during a summary computation. This behavior may change in the future if we determine that it is important to create a problem immediately.

Other RCU-Related Statements

This category both expands the original design and changes some of its behavior. The original design was discussed in Section 4.2.2. The transformers for this category now support statements that free memory. When it comes to changes, the RCU dereference does not longer create a precondition, because this member does not exist anymore. Similarly, situations where a problem can be resoled are now determined solely based on the type of the problem itself because there are no precondition. There is one more expansion, all of these transformers now add statements to the „Function Calls“ part of the abstract context. The updated transformers are as follows:

- *RCU dereference*: check if it is called within a corresponding critical section; if it is not, create a problem.
- *RCU reader synchronization*: check if it is invoked within a corresponding critical section; if it is, create a problem.
- *Deprecated and removed RCU statements*: create a problem.
- *Free and Kfree statements*: check if there are both a RCU dereference and a RCU reader synchronization primitive in the „Function Calls“ member; if not, create a problem.

As discussed in Chapter 2, RCU needs to synchronize with the readers before it can safely free the outdated memory. The purpose of the new transformers is to verify whether this happened. It checks whether the „Functions calls“ member contains the RCU reader synchronization primitive. If it does not, there are two possible reasons. It is either an error or the free statement is not related to RCU at all. If we created a problem for each situation where a free statement is not accompanied by a RCU synchronization primitive we would get a lot of false positives. For example, if we analyzed an application that does not use RCU at all, but uses several free statement, we would get an error for each one of these free statement. And of course none of these errors would be identified correctly. To combat this, we wanted to determine if the analyzed function uses RCU before creating this type of problem. It is done by check whether the analyzed function contains an RCU dereference statement in its abstract domain. We may miss some of the errors if the RCU updater does not use the RCU dereference statement but we eliminate all situations where this problem was identified incorrectly.

Other Statements

The new category includes all the other statements not included in the first two groups. The transformer for them is very simple, it just adds these statements to the „Functions Called“ member of the functions abstract context.

5.1.3 Operators

The implemented operators were changed to reflect the changes to the abstract context.

Join operator

The join operator needs to combine all four parts of the abstract context. The actions are as follows:

- *Problems*: create a new set containing all problems from both sets.
- *Postconditions*: create a new set, where for each lock determine its lock score based on the following conditions: (reasons for this conditions are discussed below)
  - if there is only a single lock score, take its score.
  - if there are two lock scores, compare them and take the higher „Maximal score“ and the lower „Minimal score“.
- *Function Calls*: create a new set containing all function calls from both sets.
- *Process Description*: take process description from either abstract context.

We decided to combine all the problems and function call from both of the abstract contexts into one set, because we do not want to miss out on any of them. When it comes to the process description, both abstract contexts should hold the same one, meaning that either can be taken. We take it from the abstract context that comes as the first operand to the join operator.

The postconditions need to be combined together too, but it cannot be done just by simply taking all of the elements from both abstract contexts and putting them into a single one. The different lock scores need to be taken into account and it is done by taking the worst case scenario from the two values. Thanks to this fact, no locking errors should be missed. However, it can introduce falsely identified errors, because the worst case scenario may not be possible to be reach.

Widening operator

The widening operator still consists simply of calling the join operator because we did not determined it to cause problems. If this fact changes, we will most likely put an upper bound on the amount of times the join operator can be used and after over-approximate the result.

Less or equal operator

This operator just checks if all postconditions, problems and functions calls of one abstract context are contained in the second one. If it is the case, the first context will be taken as the lesser one. And once again if there is even a single element that is unique for both of these abstract contexts, it means that they cannot be ordered because they are different.

5.2 Analysis Logic

The Analysis Logic also underwent major changes in all three categories. The main reason for this changes were wrong assumptions during the original design of the analysis.

Overall Logic

When we designed the overall logic, we assumed that because of Infer.AI core principles it would be sufficient to run the analysis just on an entry point function for the given source code, which would result in all the relevant functions to be analyzed too, because they need to be directly or indirectly called from this entry function. This assumption turned out to not be correct because Infer.AI does not always behave as expected.

For example the `pthread_create(...)` statement, used to create new threads, takes a pointer to a function as a parameter. This function will be called by the newly created thread, but Infer.AI does not dive into this function. It means that if we analyzed only the entry point function, then all of the functions that were called by the thread both directly or indirectly may not be analysed. As a result, the analysis may fail to detect some of the errors.

The new logic divides the analysis into two parts. Firstly, the analysis is started for all of the functions declared in the source code. This does not mean that the Infer.AI does no longer dive and analyze the called functions once it encounters them. It still does, we just make sure that no function is skipped during the analysis. Furthermore, Infer.AI guarantees that no function will be analyzed twice, because summary of every analyzed function is saved in a database. The *first part* has four steps that are repeated for every function declared in the source code:

1. *Check if the function already has a summary*, if it does, skip rest of these steps.
2. *Start the analysis* of the function, which contains the following actions:
   - *Apply Abstract Transformers* for the functions statements.
   - Let Infer.AI *dive into the called functions and analyze them.*
   - *Create summaries* of the called functions after their analysis finishes.
   - *Apply summaries* of these called functions in the callers.
3. *Finish the analysis* of the function
4. *Create a summary* for the function

After this first part, the analysis has summaries for all of the declared functions, meaning that the second part may start. The second part needs to determine which function are „top-level“ or without a caller and report errors from all of them. The *second part* has these steps:

1. *Get list* of the functions that were declared in the source file.
2. *Get summaries* of all functions in the list.
3. *Add* all function calls saved in the summaries into a single all function calls set.
4. *Decide* which functions from the list should be added to a new top-Level function. list according to the following conditions:
    - If the function *is a member* of the all function calls set, do not add it.
    - If it *is not a member*, add it to to top-Level function list.
5. *Report* problems from summaries of all the functions in the top-level function list.

Summary computation

The Summary computation has been expanded but still remains simple. It contains just two actions that are as follows:

- Check if the problems related caused by unended critical section have been resolved; it they were, remove them.
- Check if the current critical section is still active; if it is, create a problem.

Both of these situation can be checked at the end of the function, because the analysis holds all the necessary information. If the analysis finished with properly ended critical sections, meaning that both the Minimal and Maximal lock scores were equal to zero, it is safe to say that the previously unended critical sections did not cause issues for the caller, so the related Problems can be removed.

If on the other hand the caller is faced with a situation, where the critical section is still active at the end of the analysis, it creates a Problem. As a consequence, there may be multiple unended critical section errors in the callers Problem Set, each representing a different function, where this problem was encountered. Multiple errors are kept, because we cannot say for sure that the unended critical section error in the caller was caused by the called function. There may as well be an error in the caller, who should have resolved this problem. There is an another advantage of keeping multiple related problems. It makes it easier to follow how the problem traveled throughout the product.

Applying summary

The process of applying summary was expanded and adjusted to fit the new abstract context as was the case with multiple other parts discussed above. The expanded steps of *Applying summary* are as follows: (Discussed a bit more below.)

1. Check which problems are resolved and can be removed.
2. Add the unresolved problems into the callers abstract state.
3. Add the function calls of the function into the callers abstract state.
4. Update the callers abstract state according to the callers postconditions.

The caller needs to first check which problems were resolved, so it does not add them into its own abstract state. On the other hand, the unresolved problems need to be propagated to the caller so they can be reported to the user at the end of the whole analysis. As discussed earlier, the problems are reported only for the functions that do not have a caller, which is obviously not the case for the function whose summary is being applied. It is of course not guaranteed that they will be reported to the user, because they may be resolved by a different caller.

When it comes to function calls, the caller should add the one used in the function, because there is one situation, where it is crucial. It concerns the „Missing RCU reader synchronization“ error. It is possible that the synchronization with RCU readers happens in this called function. If the function calls were not propagated to the caller, the analysis may think that the synchronization did not happen at all, because the caller does not contain any such call in its function calls set.

Lastly, the RCU state of the caller needs to be updated to reflect the functions actions. It is done by adding the lock scores for each lock contained in either or both sets. If for example, the called function declared that a lock has minimal score equal to zero and maximal score equal to one, and the caller declared that the lock has both minimal and maximal score equal to one, the final lock would have minimal score equal to one and maximal score equal to two after applying summary. It would mean that there is a worst case scenario, where the critical section is started twice after the call to this function.

Chapter 6

Experimental Evaluation

This chapter focuses on the experimental evaluation of our analyzer, where it tests its scalability, soundness and completeness.

6.1 Experiments

We have experimentally evaluated our analyzer on (i) handmade and real user-space examples and (ii) the Linux kernel.

Handmade and real user-space examples

Our handmade and real user-space examples constitute a test suite containing twenty-five simple programs, where fifteen of them contain one or multiple violations of RCU rules and the other ten are used to verify that our analyzer does not detect violations in situations when there are none. Our analyzer supports both the Kernel an the User-space variants of RCU. But, we decided to create these examples using the User-space variant, because it was easier and it also allowed us to verify our analyzer on both variants as the the handmade examples use the User-space variant and the Linux kernel uses the kernel variant.

The examples containing defects were created by us to test the analyzers' ability to detect violations in situations where the they are contained within a single function and situations where they are spread across multiple functions. There is at least a single test corresponding to each of the error pattern we identified. The patterns that are classified as warnings have one example each. Each pattern that is classified as an error has one example where it is the only defect, and it is also present in examples containing several defects from various error patterns.

As for the examples without a defect, they are taken from the user-space RCU official repository[1] and they are used to test whether the RCU state is properly propagated across functions. For example, some of them focus on using wrappers because wrappers represent situations where a function either leaves an active critical section at the end of the function or situations where a critical section is ended without being started. These examples act as a control group because there is a need to verify that our analyzer does not report errors randomly.

Tests on these handmade and real user-space examples resulted in twenty-four examples analyzed correctly and a single example analyzed incorrectly. The one incorrectly analyzed

[1] https://github.com/urcu/userspace-rcu

example contains a problem corresponding to the „Mismatch of RCU flavours" error pattern. It is an expected result, because our analyzer does not support detection of warnings based on this pattern in the current version. It is our goal to add support for this pattern in the following update.

The handmade examples usually contained around a hundred lines of Code and some of the read examples contained even higher hundreds of lines of Code.

The Linux Kernel

Verifying our analyzer on the Linux kernel proved to be challenging because several problems were encountered. The first one is connected with the way Infer compiles source files. In particular, Infers' infrastructure relies on `Clang` and a custom Infers' plugin for this compiler for source file compilation. Infer requires compilation of files that should be analyzed to gather data that are needed for the analysis. It does not require them to be compiled into executable files, it just needs them to be complied into object files. The problem is that the Linux kernel is not meant to be complied with `Clang`, but rather with the `GCC` compiler. There is some support for `Clang`, but it usually requires the newest version of this compiler and even then it might be problematic.

That is where a second problem was encountered. The latest Infer release at the time when we were trying to analyze the Linux kernel was `v1.1.0`, which supports `Clang 11.1.0`. However, at this time there already was the `13.0.1` version of this compiler. Since we were analyzing the latest Linux kernel at that time, which was the `5.16-14` version, it already required the higher version of `Clang` for successful compilation. We even tried to get an experimental build of Infer, which supports the latest `Clang`. However, this attempt was unsuccessful, because the experimental build was labeled as „failing", which proved to be true as it failed to compile.

So, we tried to compile the Linux kernel with what was available. `Clang` was selected as the desired compiler in the kernels' `Makefile`, but some issues were still encountered. The first one was related to compilation flags that were not supported by `Clang`. Thankfully, we found out a way to blacklist these unsupported flags, where each flag had to be blacklisted separately.

However soon after, another problem was encountered, the compiler was unable to compile any files that contained `Assembler` code because it did not support the used syntax. We were unable to make the compilation work for these files. As a result, we decided to skip them by passing the `„-keep-going"` flag to the kernels' `Makefile`, which forced the compilation to continue even after a file failed to be compiled.

This proved to be a winning combo that allowed `Clang` to compile a significant portion of the Linux kernels' source files. The compilation itself did not finish successfully because it could not deal with all of the skipped files. However, Infer does not mind because it already captured data for the files that were compiled successfully into object files. This method captured successfully 2,557 out of 2,710 files, which represents around 94.35 % success rate.

However, using the `GNU find` and `wc` utilities, it was determined that there are 30 700 total source files in the entire Linux kernel. And less than one tenth of them was used for this particular Linux kernel compilation. The selection of which source files will be used is based on the Linux kernel configuration, which can be used for our benefit. With proper knowledge, it should be possible to create a configuration in such a manner, where most of the compiled source files use RCU. It can result in a less resource-demanding analysis because resources will be spent more wisely.

Now that the source files had been captured, it was time to analyze them. However once again, a problem was encountered. The analysis found thousands of problems, which it tried to report on the command line. Unfortunately, as it turned out, Infer was not build to report thousands of messages at such a rapid rate, which resulted in Infer to become stuck and not responsive after around a thousand of them. For this reason, we were forced to change the way Infer reports defects to users. It was changed to create an Error Log, which is a file containing defects, for each source file separately. With this change the analysis was able to finish successfully, but the Error Logs had their shortcomings too.

They made it harder to determine which source file contained the errors because the file names of the Error logs were composed of the analyzed source file name and a random `ID`. It means that there was no information about a *path* to these analyzed source files. It was especially problematic because some source files have the same file name. Moreover, the error messages in the Error logs themselves had a problem because they contained either a series of unreadable characters or information about Includes, which were only then followed by the desired error message. However, we were unable to change these error messages because the first part of the message is generated by Infer internally.

Nevertheless, on the positive side, we have successfully captured defects that can be analyzed. It proved to be a complex task because there are 1,852 Error logs each containing somewhere from ten to several hundreds of error messages. Moreover, we came to a realization that the errors themselves are reported for a pre-processed version of the source files and not the original version of source files because various errors were reported for functions that are not present in the original source file or for lines of code that are empty in the original source file.

As a result, we tried to obtain the pre-processed versions of these source files. We tried running the Linux kernel compilation with the `-save-temps` flag that saves the temporary files during the compilation, where the pre-processed version of the compiled files is one of them. However, this attempt failed because the use of this flag introduced errors to the kernels' compilation that could not be skipped because the compilation did not allow it.

Then we came up with an idea to try and pre-process these files manually, but this plan was postponed until a way will be found that will automate this process, because it would require a significant time investment. As a result, the analysis of the Linux kernel cannot be labeled as successful just yet. However, we managed to lay significant building blocks for the analysis to work in the future.

6.2 Evaluations of the Current State and Future Plans

Thanks to the handmade examples we were able to verify that our analyzer is capable of detecting most of the situations when RCU is used in a different manner than it was designed to be. Moreover, the evaluation on the Linux kernel showed us that our analyzer is capable of analyzing even a much more complex application composed of thousands of files. However, our analyzer has some shortcomings too. Reporting of the found defects to the user is problematic for larger applications. Furthermore, the analysis of the Linux kernel found thousands of defects, which is a rather large number that can mean that a significant portion of these defects are duplicates or false positives. It would however, require an analysis to determine what is the exact cause.

As a result we plan on making several updates to the analyzer, like improving and expanding the analyzers support for error patterns defect detection, increasing both the completeness and soundness of the analysis and making the analysis practical on the Linux kernel. When it comes to error patterns, we would like to add support for at least the missing „Mismatch of RCU Flavors“ error pattern that is already identified. Moreover, we would like to analyze real RCU code to better identify how it is used, which may result in new error patterns that can expand our analyzers functionality and with it its usefulness. For soundness and completeness, we would like to determine the causes of false positives and miss the errors. Then we plan to change the analyzer to eliminate these causes to make our analyzer more reliable.

We also plan to continue with the evaluation of our analyzer on the Linux kernel, where we want to make the analysis work on any kernel version. We plan on reworking the reporting portion of our analyzer to make the error messages easy to understand and to make it easier to find these defects in their source files. Furthermore, we would like to introduce a solution that would create pre-processed version of compiled source files for the developers to analyze after the analysis finishes. There are also some overall improvements that we would like to make to the analyzer. We plan on redesigning the abstract context and transformers to be more effective by removing all unnecessary information that is kept by the analyzer during the analysis.

Chapter 7

Conclusion

The goal of this thesis was to study the principles of synchronisation of concurrent threads using RCU and using this knowledge to design and implement an analyser in the Facebook Infer framework targeted at discovering synchronisation errors in programs using RCU.

To satisfy this goal we had to overcome several challenges along the way. The first step was to get familiar with Read Copy Update to understand how this synchronization mechanism works. This step was important to understand what types of errors may happen during the RCU synchronization and what is their typical cause. Therefore, our main focus was to determine how RCU needs to be used while writing an application for its synchronization to work as desired.

The second step involved researching available solutions for error detection involving RCU and getting familiar with both the abstract interpretation and the Facebook (now Meta) Infer framework. This step was necessary to make sure that we were not „reinventing a wheel“ or rather creating and already existing and used analyzer. It was also needed to get familiar with the principles of abstract interpretation so we would be able to design the analysis around it, the same can be said for Infer.

The third step focused on the original design of our analyzer that involved two separate parts. We introduced error patterns that we identified and which are used for the error defection by out analyzer. Secondly, we discussed how we imagined our analyzer to work. The fourth step then focused on the real implementation of our analyzer, where we showed how and why we were forced to change the original design. Here, we also discussed the changes that expanded our analyzer to produce more accurate results. The last very important step focused on the experimental evaluation of our product, where we proved that it is capable of analysing even complex real-life applications.

We managed to create an analyzer that can be used to detect synchronisation errors based RCU not being used as it was intended to be. It provides support for all possible RCU flavors in both user-space and the Linux kernel. Furthermore, it was proven to work on smaller both handmade and real examples and it also managed to analyse the Linux kernel despite numerous challenges along the way. An even more improved support for large and complex applications is one of our main goals for the future. We also hope that it may become a part of either the Linux kernel or user-space RCU projects development.

Bibliography

[1] *About Infer* [online]. [cit. 2022-05-03]. Available at: https://fbinfer.com/docs/about-Infer.

[2] *Building checkers with the Infer.AI framework* [online]. [cit. 2022-05-05]. Available at: https://fbinfer.com/docs/absint-framework/.

[3] *Pulse* [online]. [cit. 2022-05-06]. Available at: https://fbinfer.com/docs/checker-pulse/.

[4] *RacerD* [online]. [cit. 2022-05-06]. Available at: https://fbinfer.com/docs/checker-racerd/.

[5] *Starvation* [online]. [cit. 2022-05-06]. Available at: https://fbinfer.com/docs/checker-starvation/.

[6] COMMUNITY, T. kernel development. Coccinelle. [online]. [cit. 2022-04-24]. Available at: https://www.kernel.org/doc/html/latest/dev-tools/coccinelle.html.

[7] COMMUNITY, T. kernel development. Development tools for the kernel. [online]. [cit. 2022-04-24]. Available at: https://www.kernel.org/doc/html/latest/dev-tools/index.html.

[8] COMMUNITY, T. kernel development. The Kernel Concurrency Sanitizer (KCSAN). [online]. [cit. 2022-04-24]. Available at: https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[9] COMMUNITY, T. kernel development. Kernel Electric-Fence (KFENCE). [online]. [cit. 2022-04-24]. Available at: https://www.kernel.org/doc/html/latest/dev-tools/kfence.html.

[10] COMMUNITY, T. kernel development. Kernel Testing Guide. [online]. [cit. 2022-04-24]. Available at: https://www.kernel.org/doc/html/latest/dev-tools/testing-overview.html.

[11] COMMUNITY, T. kernel development. *Lockdep-RCU Splat* [online]. [cit. 2022-04-28]. Available at: https://www.kernel.org/doc/html/latest/RCU/lockdep-splat.html.

[12] COMMUNITY, T. kernel development. *RCU and lockdep checking* [online]. [cit. 2022-04-28]. Available at: https://www.kernel.org/doc/html/latest/RCU/lockdep.html.

[13] COMMUNITY, T. kernel development. Runtime locking correctness validator. [online]. [cit. 2022-04-24]. Available at: https://www.kernel.org/doc/html/latest/locking/lockdep-design.html.

[14] COMMUNITY, T. kernel development. Sparse. [online]. [cit. 2022-04-24]. Available at: https://www.kernel.org/doc/html/latest/dev-tools/sparse.html.

[15] COMMUNITY, T. kernel development. *What is RCU? – "Read, Copy, Update"* [online]. [cit. 2022-04-16]. Available at: https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html.

[16] COUSOT, P. Semantic Foundations of Program Analysis. In: MUCHNICK, S. and JONES, N., ed. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981, chap. 10, p. 303–342 [cit. 2022-05-03].

[17] COUSOT, P. Interprétation abstraite. *Technique et science informatique*. Paris, France: Hermès. january 2000, vol. 19, 1-2-3, p. 155–164, [cit. 2022-05-03].

[18] COUSOT, P. Abstract Interpretation Based Formal Methods and Future Challenges, invited paper. In: WILHELM, R., ed. *« Informatics — 10 Years Back, 10 Years Ahead »*. Springer-Verlag, 2001, vol. 2000, p. 138–156 [cit. 2022-05-03]. Lecture Notes in Computer Science.

[19] COUSOT, P. Abstract Interpretation in a Nutshell. [online]. [cit. 2022-05-03]. Available at: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html#CousotAIMIT.

[20] DESNOYERS, M., DAGENAIS, M. R., WALPOLE, J., MCKENNEY, P. E. and STERN, A. S. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel & Distributed Systems*. Los Alamitos, CA, USA: IEEE Computer Society. feb 2012, vol. 23, no. 02, p. 375–382, [cit. 2022-04-16]. DOI: 10.1109/TPDS.2011.159. ISSN 1558-2183.

[21] DESNOYERS, M. and MCKENNEY, P. E. *Userspace RCU* [online]. [cit. 2022-04-21]. Available at: https://liburcu.org/.

[22] HARMIM, D. *Static Analysis Using Facebook Infer To Find Atomicity Violations*. Brno, 2019. [cit. 2022-05-03]. 5-7 p. Bachelors Thesis. Brno University of Technology, Faculty of information technology. Supervisor ING. TOMÁŠ VOJNAR, P. prof. Available at: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=197619.

[23] LENGÁL, O. and VOJNAR, T. *Abstract Interpretation: Ingredients of Abstract Interpretation* [online]. Presentation. Brno University of Technology, Faculty of Information Technology, 2022 [cit. 2022-05-03]. 6-12 p. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-06.pdf.

[24] MCKENNEY, P. E. *RCU* [online]. [cit. 2022-04-24]. A crossroad to most of the RCU related sources maintained by one of the RCU founders Paul E. McKenney. Available at: http://www2.rdrop.com/users/paulmck/RCU/.

[25] MCKENNEY, P. E. What is RCU? Part 2: Usage. *LWN.net*. Eklektix, Inc. december 2007, [cit. 2022-04-06]. Available at: https://lwn.net/Articles/263130/.

[26] MCKENNEY, P. E. Sleepable Read-Copy Update. october 2009, p. 2–4, [cit. 2022-04-20]. Available at: https://www.researchgate.net/publication/255592374_Sleepable_Read-Copy_Update.

[27] McKenney, P. E. The RCU API, 2010 Edition. *LWN.net*. Eklektix, Inc. december 2010, [cit. 2022-04-28]. Available at: https://lwn.net/Articles/418853/.

[28] McKenney, P. E. A Critical RCU Safety Property Is... Ease of Use!!! In: IBM Corporation. [online]. May 2019, p. 25,29 [cit. 2022-04-24]. DOI: 10.1145/3319647.3325836. Available at: http://www2.rdrop.com/users/paulmck/RCU/rcu-exploit.2019.06.05d.pdf.

[29] McKenney, P. E. The RCU API tables, 2019 edition. *LWN.net*. Eklektix, Inc. january 2019, [cit. 2022-04-20]. Available at: https://lwn.net/Articles/777165/.

[30] McKenney, P. E. Does RCU Really Work?: And if so, how would we know? In: Facebook Corporation. [online]. March 2021, p. 23 [cit. 2022-04-20]. Available at: http://www.rdrop.com/users/paulmck/RCU/Validation.2021.03.24a.pdf.

[31] McKenney, P. E. Unraveling RCU-Usage Mysteries: Additional Use Cases. In: Facebook Corporation. [online]. February 2022, p. 12 [cit. 2022-04-16]. Available at: https://events.linuxfoundation.org/wp-content/uploads/2022/02/RCUusageAdditional.2022.02.22b.LF-1.pdf.

[32] McKenney, P. E. and Walpole, J. What is RCU, Fundamentally? *LWN.net*. Eklektix, Inc. december 2007, [cit. 2022-04-17]. Available at: https://lwn.net/Articles/262464/.

[33] Møller, A. and Schwartzbach, M. I. *Static Program Analysis*. February 2022 [cit. 2022-05-02]. Department of Computer Science, Aarhus University.

[34] Techopedia. *Torture Test* [online]. [cit. 2022-05-10]. Available at: https://www.techopedia.com/definition/16336/torture-test.