THE ESBMC-BASED APPROACH OF SECURITY VERIFICATION IN LINUX KERNEL PROGRAMS

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

Student id: 11050942

Department of Computer Science

Contents

Word Count: 13083

List of Tables

List of Figures

Abstract

THE ESBMC-BASED APPROACH OF SECURITY VERIFICATION IN LINUX KERNEL PROGRAMS
Zhicheng Zhou
A dissertation submitted to The University of Manchester
for the degree of Master of Science, 2023

The security verification of the Linux kernel has always been a widely concerned issue. One reason for this is that the Linux kernel exists in many important systems, and its security vulnerabilities can cause many immeasurable security problems. Another reason is that the security of the Linux kernel has never been easy compared to the detection of user-mode programs. The Linux kernel is very low-level and involves many complex dependencies and hardware. How to efficiently and effectively conduct security verification for Linux kernel programs is a challenging topic. This project models the operations of the core by simulating the functions of the kernel, to check whether the program meets the conditions it should meet. Instead of dynamic verification, this project uses bounded model checking for verification. As mentioned earlier, the Linux kernel program's dependencies are very complex, and this project establishes simplified dependencies in the ESBMC code base by simulating the kernel source code, to more smoothly abstract and model the core operations of kernel functions. This project successfully tested the memory management of the linux kernel through simulation and also realized the prevention of race condtion in kernel mode.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Professor Lucas Cordeiro. He provided me with very patient and detailed guidance throughout the research of the project. He helped me with the issues I met during my research, and asked other PhD students to give me some suggestions.

Chapter 1

Introduction

The complexity of software development is constantly increasing, especially in the area of system-level programming. Systems with a few thousand lines of code in the past have developed into massive architectures with millions of lines today. As the scale of the program is growing up, the difficulty of maintaining the codes and guaranteeing the security of the program arises. The operating system is the core component of these systems since it makes sure everything functions properly, and schedules the collaborations between hardware and user programs. The Linux kernel stands out among the top operating systems due to its open-source status and wide use. But because of its extensive code base and importance in reconciling hardware and software elements, it is a key area of reliability and security concern. The robustness and security of the kernel have to be guaranteed so that the system would not crash due to any potential vulnerability.

The importance of kernel verification for the industry is highlighted by the growing integration of Linux-based systems in crucial applications, from mobile systems, automobile control units to even medical equipment. Any security or dependability flaw in the kernel could lead to different degrees of consequences. Industrial experts have committed significant resources to kernel verification as a result of realising this, encouraging partnerships with the academic community and supporting improvements in verification methods. The techniques under the field of formal methods, static analysis and bounded model checking, are developed and utilized to solve the concern of the potential vulnerabilities in Linux kernel programs.

Even if they are useful, conventional methods of software development typically miss tiny Linux kernel bugs. Although extensive testing and code reviews are important, they are not always successful in revealing hidden flaws and minor errors that could cause system instability or security breaches. Formal verification methods provide an alternative by providing mathematical evidence of program correctness. EMBMC has distinguished itself as a useful tool for comprehensive analysis of big software systems among these methods.

When applied to systems as complex as the Linux kernel, even formal verification methods like ESBMC have difficulties. Because of the kernel's size and the vast range of hardware configurations it supports, verification tools face scalability and complexity challenges. Furthermore, replicating real-world scenarios in a verification environment might be difficult, limiting the extent of faults that can be found.

1.1 Problem Description

The Linux kernel serves as the operating system kernel for a significant majority of servers, smartphones, and embedded systems. Given its integral role and widespread adoption, the Linux kernel has a vast codebase, teeming with complex functionalities and interactions. The dependencies between each component, or code file are also nested deeply. This complexity, while allowing for rich features and adaptability, also causes potential vulnerabilities, especially when the user programs involve some particular functions with kernel codes. These functions are designed to perform specific tasks. However, their behaviour can vary based on the calling program's state and input parameters. A misalignment or a mistaken assumption about a kernel function can inadvertently introduce vulnerabilities. Such vulnerabilities can lead to system malfunctions, memory corruption, and, more worryingly, exploitable security breaches.

The paper from[2],*'Linux kernel vulnerabilities: State-of-the-art defences and open problems'* also provides an overview of this topic. This study classifies 141 Linux kernel vulnerabilities published in the CVE list between January 2010 and March 2011 and classifies the types of attacks that can exploit these vulnerabilities. It states that most vulnerabilities fall into 10 categories, e.g. Missing permission checks, Buffer Overflow, and Memory Mismanagement, based on the type of programming error developers make. This study also shows the distribution of vulnerabilities in the Linux kernel source tree. Some state-of-the-art prevention is introduced, but not deeply investigated.

This research is based on the Bounded Model Checking(BMC), and more peculiarly, the Bounded Model Checker ESBMC (the Efficient SMT-based Context-Bounded Model Checker). By achieving first place in the ReachSafety-XCSP subcategory and second place in the SoftwareSystems-AWS-C-Common-ReachSafety, ReachSafetyECA, and ReachSafety-Arrays subcategories, ESBMC has demonstrated excellent efficiency and outstanding error detection ability as a state-of-the-art BMC tool[3]. The mechanism of how the kernel handles memory management, e.g., dynamically allocating the memory, and process multi-threads programs are fully explored in the following sections.

The project aims to answer the research question in the following aspects:

1. **Abstract kernel functions and mock dependencies:** Design a methodology to abstract the core operations of kernel functions and mock the inherent dependencies. This aim is to create a representative model of the Linux kernel's behaviour, which, while simplified, retains the essential characteristics for accurate verification.

2. **Develop robust and reasonable verification mechanisms for verifying kernel codes based on ESBMC :**Focus on the implementation that utilizes ESBMC's capability with newly-developed modules to detect the vulnerabilities in Linux kernel codes.

3. **Optimize the verification by generating comprehensive regression tests:** Improve the efficiency and accuracy of the verification process by testing with well-designed and comprehensive tests. When the user improperly writes a kernel program, the potential risk should be detected.

1.2 Objectives

The objective of the project is to implement the additional functionality of ESBMC to detect the vulnerabilities in Linux Kernel codes, e.g. kernel function calls. The objective of the project is focused on the verification of memory management and concurrency in kernel calls.

1. **Integrate ESBMC with Kernel Code Analysis:** Incorporate ESBMC's bounded model checking capabilities into the verification tool, ensuring it can navigate and analyze the intricate landscape of Linux kernel codes effectively.

2. **Implement Dependency Mocking and Operation Abstraction:**Implement the related dependency, e.g. header files of the verified kernel functions in the ESBMC library, and model the kernel functions by mocking the core operations.

3. **Evaluate the implementation and refine:**Once the verification approach is developed, conduct rigorous testing on a variety of kernel interfacing programs. Identify the expected outcomes and gather the results to iteratively improve the design of modelling

1.3 Contributions

As mentioned in the earlier sections, the problem this project aims to resolve is the verification of potential defects in Linux Kernel programs. The project delivered a solution based on modelling the verification by mocking the core operations of the kernel functions. The contributions of this project are listed:

1. The project provided a distinct approach to verify the Linux Kernel programs. It mocked the complex and deeply-nested dependency of Linux Kernel source codes and simplified the dependency. In order to include the native Linux Kernel header files while using ESBMC to parse the program, this project made the mocked dependency as the kernel libraries in ESBMC source codes. The library*c2gogo/headers/ubuntu20.04/kernel_5.15.0-76* was created for this purpose. This project bundled this library into ESBMC for kernel related operations. The mocked header files in this library contains the corresponding liscence from the specific version Linux Kernel source codes.

2. The project delivered a robust solution targeting on the memory management in Linux Kernel. It applied the modelling on dynamic memory allocation in Linux Kernel. The mocked functions ,*kmalloc*, *kfree*, *kmalloc_array*, and other related ones, achieved reasonable and effective verification on the arguments used for allocating the memory in kernel. For example, the potential vulnerability that negative allocating size(passing negative value into an unsigned int argument leads to very huge positive number) was perfectly detected and verified by this project. The inspection on the flags for allocating memory was also delivered. The project also handled the issue of memory leak in Kernel properly.

3. This project also implemented verification on data transmission between kernel

space and user space. It modelled the operations related to kernel behaviors that copying from kernel space to user space, and vice versa.

4. Besides memory management, the project delivered a adequate solution for detecting the race condition in concurrency. It modelled the *spin_lock* related operations to simulate the kernel-level functions in user space. Besides the implementation, the contributions of this project also contains a number of regression tests to successfully evaluate if the approach works in an expected way.

1.4 Dissertation Structure

The structure of the dissertation is divided into 5 chapters. The first chapter is a brief introduction to the project and the research field the project lays on. In this chapter, the topic of this project is introduced, including the problem description, objectives and the contributions of this project.

Chapter 2 mainly focuses on the background of this project. It contains the background knowledge related to this project and the related works. It is divided into 2 sections:Linux Kernel and Verification Techniques. The Linux kernel, which serves as a cornerstone of our research, is discussed in detail in the first section. Given the kernel's widespread importance in the world of operating systems and its complex operations, a detailed understanding of it is essential. This section is also crucial to this project since the modelling of the kernel behaviours relies heavily on the background knowledge of the kernel. The subsequent section pivots to the domain of model checking, This is fundamental to our scientific approach and is essential for confirming system accuracy. The subtopics involved in this section are Static Analysis, Bounded Model Checking(BMC), satisfiability modulo theories(SMT) solvers and the software that supports this research approach, ESBMC. The related works are introduced in this section to provide a more comprehensive overview of the research question.

Chapter 3 focuses on the methodology of the project. It first identifies the gap, or challenge to implementing the verification techniques for the Linux kernel, and then deeply analyzes and demonstrates the approach or solution to the challenge. This chapter explains all modellings of different kernel behaviour in detail, including the design, process, and pre and post-conditions that needed to be verified. Several diagrams are listed in this chapter to support the illustration.

Chapter 4 is the section for evaluation. The regression tests and benchmarks for testing are fully discussed. The benchmark suite is explained and demonstrate with tables. The results of the evauatation are also discussed in this chapter.

Chapter 5 summarizes the deliverables and significant achievements of this project, and indicates the limitation and the potential future work that could be applied.

Chapter 2

Background and Theory

This chapter is dedicated to providing a comprehensive background that sets the stage for our research's depth and breadth. This chapter not only dives into the technical details of our key areas, e.g. how Linux Kernel handles memory management, in great detail, but it also examines and discusses earlier studies in the relevant subjects, ensuring that our investigation is built on the shoulders of giants and is grounded in a thorough understanding. It is divided into two primary sections: Linux Kernel and Model Checking.

2.1 Linux Kernel

One of the most significant and well-known open-source projects in computer history, the Linux kernel was created by Linus Torvalds in the early 1990s and functions as the brains behind the Linux operating system. The fact that Linux is a free operating system is one of its more alluring features. The GNU General Public Licence (GPL) makes its source code open and available for anyone to study. If you download the code (the official site is http://www.kernel.org) or look up the sources on a Linux CD, you will be able to examine one of the most popular modern operating systems from top to bottom[4].

The kernel, instead of the whole operating system, is responsible for the core components that enable the system to behave properly. The kernel is the innermost part of the operating system, whereas the user interface is its outermost component. It is the system's fundamental components; the software that manages the hardware and allows resources across the system's various components[5]. A Unix-like operating system

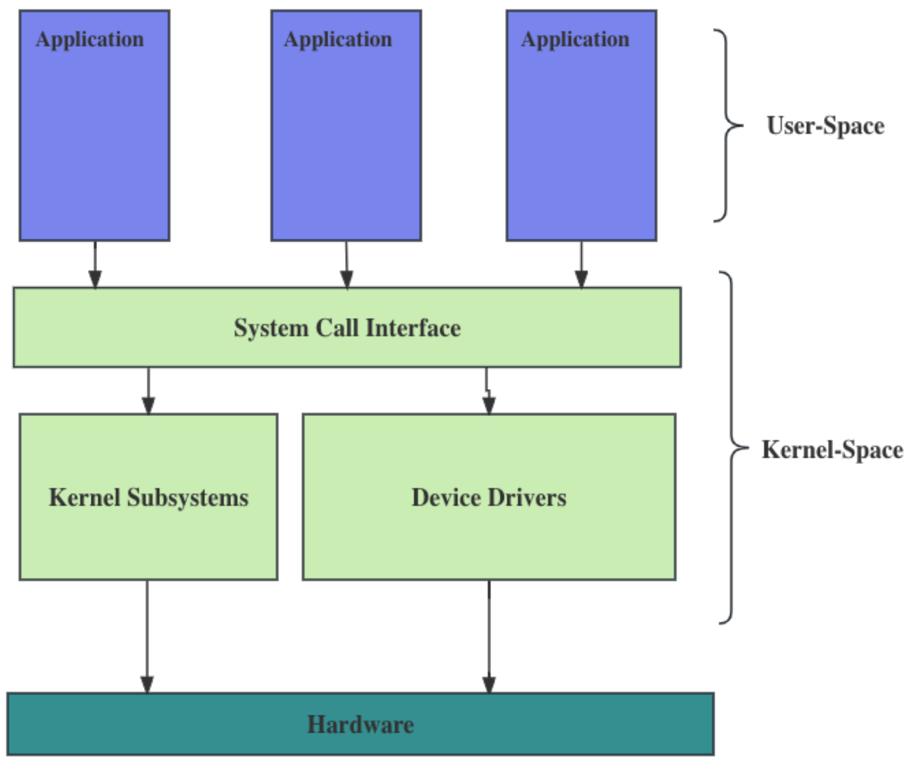

Figure 2.1: The relationships between applications, kernel and hardware

conceals all low-level information about the physical setup of the computer from user-run applications. A programme must submit a request to the operating system to utilise a hardware resource. The kernel assesses the request and, if it decides to approve it, engages the relevant hardware on behalf of the user programme[4]. Thus, the operating system is divided into user and kernel modes. This project aims to explore and prevent vulnerabilities in kernel mode. The kernel is further divided into different modules: System Call Interface(SCI), Process Management(PM), Memory Management(MM), Virtual File System(VFS), Network Stack and Device Drivers(DD). Each module handles the corresponding kernel function calls.

Figure 2.1 illustrates how the kernel, user program and hardware are involved in cooperation. The user program must involve kernel function calls to schedule any particular hardware. Figure 2.2 lists the core modules in the kernel. Each module takes charge of different aspects of tasks, that are significant to the proper functioning of the system. .Since the project is focusing on verifying the security issues related to kernel memory management and kernel concurrency, only Process Management(PM)

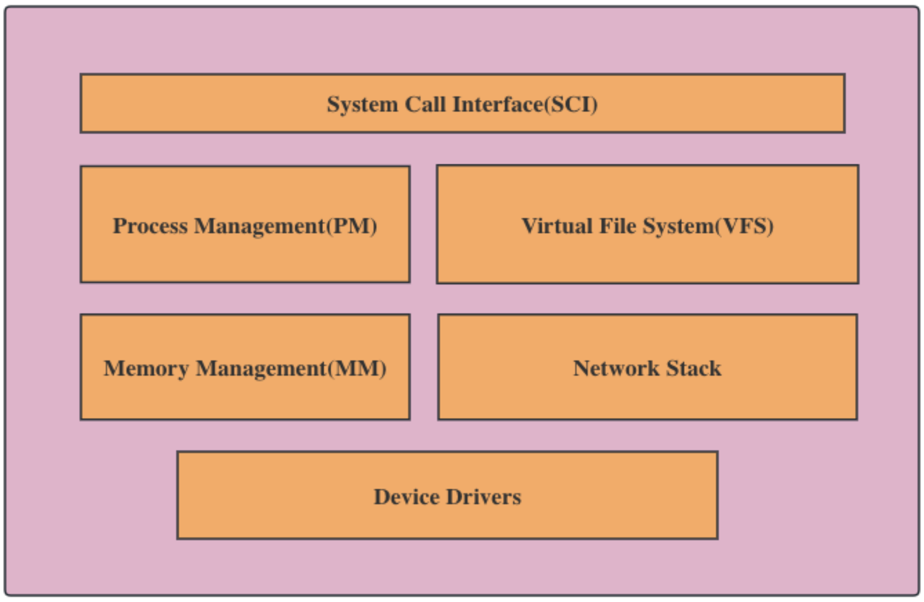

Figure 2.2: The modules of the kernel: each module handles several tasks

and Memory Management(MM) are further covered.

Process Management: This module is responsible for task scheduling, process lifecycle management, context switching between processes, and managing process states (like running, waiting, or stopping). The kernel is a process manager, not a process itself. The process/kernel model presupposes that processes will use system calls, which are specialised programming constructs when they need a kernel service. To convert from User Mode to Kernel Mode, each system call first sets up the group of parameters that uniquely identifies the process request and then executes the hardware-dependent CPU instruction[4].

Memory Management: It manages the system's virtual and physical memory. Memory allocation, paging, swapping, and memory mapping are important duties that apply to both kernel and user space.

2.1.1 Kernel Memory Management

The memory management subsystem is a crucial part. Modern computers have enormous quantities of memory, and an OS may run numerous processes at once, thus effective memory management becomes crucial to ensuring the best system responsiveness and performance. It is more difficult to allocate memory inside the kernel than outside of it. Simply, the kernel does not have the sophistication that userspace does. The kernel is not usually allowed to allocate memory quickly, unlike user-space[5]. In addition, because the activities are independent of the architecture yet are both specific to it, the overall design and implementation must be fair and flexible[6]. The main responsibilities of memory management are: allocating and freeing memory pages, managing virtual and physical memory spaces, swapping, and facilitating memory mapping. These functions ensure that each process views memory as a large, continuous space, abstracting away the underlying complexities of the physical memory layout. Before exploring the verification approach on kernel programs, several concepts related to memory management have to be addressed.

Page

The fundamental unit of memory management in the kernel is a physical page. The smallest addressable unit for a CPU is a byte or a word, but the memory management unit (MMU), which is the hardware that controls memory and translates virtual addresses into physical addresses, usually works in pages. As a result, the MMU maintains the system's page tables with a granularity of one page (thus their name). The smallest unit that counts in virtual memory is a page[5]. A page is a unit of contiguous memory with a fixed size. This size is commonly set to 4KB in the Linux kernel on many architectures, while this is not a requirement. In Linux Kernel source codes, a *struct* is used to describe the data structure of a page.
The simplified codes of *struct page* are shown below:

```
struct page {
        unsigned long flags;
        atomic_t _count;
        atomic_t _mapcount;
        unsigned long private;
        struct address_space *mapping;
        pgoff_t index;
        struct list_head lru;
        void *virtual;
};
```

The unsigned long bit field *flags* that describe the physical page's current state. In the kernel header file including/Linux/page-flags.h, an enum serves as the definition for flag constants[6]. Each flag constant indicates different states of the page. For example,

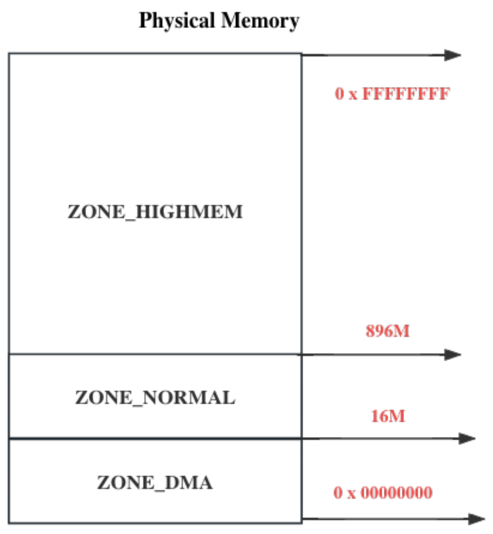

Figure 2.3: The distributions of memory zones in physical memory.

PG_locked indicates if the page is locked. When I/O operations are started on a page, this bit is set, and it is cleared when they are finished. *PG_active* indicates if the page is in the active list, and *PG_slab* shows that the slab allocator, which is introduced in a later section, is in charge of managing the page[4]. The number of references to this page is kept in the *_count* field. When this number falls below one, no one is utilising the page, and it is free to be used in a new allocation. The *virtual* field is the page's virtual address. Another important field is *mapping*, which represents a pointer of type *address_space*. Struct *address_space* is an abstraction that represents a set of pages engaged for a file cache[6]. Thus, the *mapping* field in *struct page* serves as a bridge between a memory page and the file it might be associated with.

Zone

Zones are used to address the varied needs of various memory areas, particularly in architectures with particular memory access restrictions. To achieve effective allocation based on various access needs, kernel segments physical memory. As an illustration, whereas some memory addresses are better suited for Direct Memory Access (DMA) activities, others are more appropriate for system-wide operations. During system boot, the kernel locates and divides memory into zones, optimising allocation choicesd[4]. The ”normal” page frames that the kernel can directly access through the linear mapping in the fourth gigabyte of the linear address space are included in the ZONE_DMA and ZONE_NORMAL zones. In contrast, the ZONE_HIGHMEM contains memory page frames that cannot be directly accessed by the kernel through the linear mapping in the fourth gigabyte of linear address space[4]. This memory zone was introduced to efficiently manage and use memory that exceeds this limited direct-mapped region.

Slab Allocator

The page allocator effectively handles requests for memory allocation that are multiples of page size while working with the buddy system. However, the majority of allocation requests made by the kernel code are for smaller blocks (often less than a page); utilising the page allocator for these allocations causes internal fragmentation, which wastes memory.[4]. The slab allocator is implemented to resolve this. It utilizes the object cache concept, where free page frames are reserved, segmented into slab caches, and organized into lists of free pages. Each list has a unique unit size, and its pool consists of memory blocks of that size. When the memory of a specific size is requested, the allocator identifies the best-fitting slab cache and provides a free block's address. Despite its apparent simplicity, managing slab caches is intricate, necessitating careful object tracking, dynamic expansion, and safe reclaim through interfaces like the shrinker. This balance between performance and memory footprint is complex and critical[4]. The further detailed mechanism of slab allocator is not discussed in this paper since it is not directly related to our research question. But it is necessary to address that the slab allocator efficiently manages memory by grouping blocks of fixed sizes in caches and reducing memory fragmentation.

2.2 Formal Verification Techniques

2.2.1 Static Analysis

Static analysis refers to the set of methods used to evaluate a codebase for potential errors, vulnerabilities, or deviations. Static analysis tools can identify many common coding problems automatically before a program is released. They examine the text of a program statically, without attempting to execute it. They can examine either a program's source code or a compiled form of the program to equal benefit, although the problem of decoding the latter can be difficult[7].

Static analysis was originally concentrated on Fortran and was confined to a single procedure (intraprocedural analysis). However, even this simple form of static analysis is not recursives[8] .This means that if a function or procedure called itself either directly or indirectly, this early form of static analysis would not be able to fully capture or understand the implications of such recursive calls. Due to its focus on a predetermined set of patterns or rules in the code, static analysis is unable to address every security issue. It is unable to offer design advice but can identify bugs in the finest of details. Although static analysis techniques can identify problems in small details, knowledgeable individuals still need to design a program correctly to avoid any flaws[7]. As programs grew in complexity, encompassing multi-procedure structures and recursive patterns, the limitations of the initial static analysis methodologies became evident.

The paper *A Comparative Study of Industrial Static Analysis Tools* surveys three state-of-art static analysis tools, PolySpace Verifier, Coverity Prevent, and Klocwork K7. The survey draws from research articles and manuals, examining defect types like memory management and security vulnerabilities, along with soundness, value analyses, incrementality, and IDE integration[9]

Another open-source static analysis tool for Java, FindBugs, is described in detail in the paper *Using Static Analysis to Find Bugs* . This tool is capable of recognizing more than 300 programming mistakes and questionable coding practices using straightforward analysis techniques. It identifies defects like potential null pointer dereferences, checks where code could be inconsistent in handling null values, conducts intraprocedural type analysis and detects instances where objects of guaranteed unrelated types are compared for equality[10]. Thus, static analysis stands as a powerful technique in the software development landscape, providing an automated method without the necessity for actual code execution to find potential bugs, weaknesses, and coding standards violations. However, it is inherently limited in capturing runtime behaviours and can sometimes produce false positives. The development of static analysis tools lasts a number of years and effectively improves the verification of software. Different tools may focus on different aspects of defects, but also provide beneficial foundation for newly-developed tools. The concurrency issues are also concerns that some static analyzers target.This article covers a C code static analyzer for detecting spinlock usage in the Linux kernel. Misuse of spinlocks is difficult to detect and rather widespread, resulting in runtime deadlocks in the Linux operating system kernel on multiprocessor architectures[11].Spinlocks are the kernel's preferred discretionary mutual exclusion technique, keeping other kernel threads out of a spinlocked area of code while that code is being performed.

2.2.2 SMT Solvers

Solvers aim to determine if given formulas are satisfiable. These formulas are broadly categorized into Boolean satisfiability (SAT) and satisfiability modulo theories (SMT). In the context of SAT, the main query is if a particular Boolean formula can be validated, and tools designed for this purpose are known as SAT solvers. Two primary decision-making mechanisms leveraged by these solvers are the Davis-Putnam-Logemann-Loveland (DPLL) method and the Conflict Driven Clause Learning (CDCL) approach. However, the latter, CDCL, is more favoured because of its more efficient time complexity.

One limitation of SAT is that the process of translating higher-level system designs to Boolean logic can be resource-intensive. This challenge led to the development of SMT solvers, which offer verification mechanisms capable of understanding more abstract levels while still harnessing the efficiency and automation typical of Boolean engines. In the realm of mathematics, a theory is perceived as a cohesive collection of first-order formulas, spanning domains like Equality, Bit-vector, Linear-arithmetic, and Arrays. Such theories prescribe specific rules to infer and simplify intricate formula sets. There are several contemporary SMT solvers available, showcasing the evolution of this field. The verification software applied in this project, ESBMC, uses various modern SMT solvers.

1. Yices, crafted by SRI International, is a proficient SMT solver that accommodates a diverse mix of first-order theories, beneficial for both software and hardware representation[12]. It can adeptly manage extensive and propositionally intricate formulas across a diverse array of theories. Yice accommodates all the theories outlined in SMT-LIB, encompassing uninterpreted functions, difference logic, linear arithmetic for real and integer values, extensional arrays, and bit vectors.

2. Z3, developed by Microsoft Research, is an innovative SMT solver optimized for addressing challenges in software verification and analysis tasks[13].Z3 is applied in tools such as Spec#/Boogie, Pex, HAVOC, Vigilante, VCC (a verifying C compiler), and Yogi.

3. MathSAT[14] is a versatile SMT (Satisfiability Modulo Theories) solver developed through a collaboration between FBK-IRST and the University of Trento. It's equipped to handle a wide range of SMT theories, with notable support for floating points. It is also equipped to generate Craig-interpolants and carry out partial assignment enumeration. For academic purposes, it's offered as a complimentary, non-commercial license.

4. Boolector is a proficient SMT solver that specializes in the quantifier-free theory of bit-vectors and arrays. It employs techniques like term rewriting, bit-blasting, and on-demand lemmas for array processing.[15].

5. CVC4 is the newest iteration of the Cooperating Validity Checker, a collaboration between NYU and the University of Iowa. It aims to incorporate the best features of CVC3 and SMT-LIBv2 while benefiting from recent advancements in system architecture and decision procedures[16]. Despite being a complete rewrite of CVC3 with various redesigned subsystems, CVC4 is more streamlined and outperforms its predecessor, with further enhancements in the pipeline.

2.2.3 Bounded Model Checking(BMC)

In the field of formal methods, model checking is a key tool for confirming the correctness of finite-state systems. This approach focuses on algorithmically comparing system models to desired specifications. Model checking's main appeal is its ability to give a systematic evaluation of the full state space of the system being examined, ensuring thoroughness in validation.
Model checking can be made up of several tasks. Initially, there's the "modelling" phase where the design is translated into a formal structure, making it compatible with the specificities of model checking tools[17]. Following this is the "specification" stage, where the use of logical formalisms helps in defining the properties the design ought to fulfil. The final step is "verification", wherein the design is rigorously evaluated against the previously outlined specifications to ascertain its compliance. Together, these stages offer a comprehensive review of the design, ensuring its alignment with stipulated criteria[17].

The research *Experimental Analysis of Different Techniques for Bounded Model Checking* compares the performance of SAT-based, BDD-based, and explicit state-based BMC on commercial design benchmarks. The experimental framework provides a consistent and comprehensive foundation for assessing each technique. The results indicate that BDD-based BMC is substantially faster for designs with deep counterexamples. SAT-based BMC outperforms BDD-based BMC for designs with shallow counterexamples, although explicit state-based BMC is comparable[18].

The paper *Bounded Model Checking Using Satisfiability Solving* integrates model checking with satisfiability solving, known as bounded model checking. This technique promises a more efficient exploration of the state space[19]. Thus, Bounded model checking (BMC), on the other hand, is a specialized variant of model checking. BMC verifies the system's attributes only up to a specific predefined depth or bound, unlike classical model checking, which thoroughly investigates all states. Moreover, it is a method for the detection of logical errors in finite-state transition systems, positioning BMC as an alternative approach to symbolic BDD-based model checking[20]. BMC operates by formulating a propositional formula that becomes satisfiable when a specific path is identified. Different designs of bounded model checking techniques may vary in their approach, effectiveness, and computational efficiency. The study *Experimental Analysis of Different Techniques for Bounded Model Checking* evaluates the efficiency of SAT-based, BDD-based, and explicit state-based bounded model checking (BMC) using benchmarks from commercial projects. Through a consistent experimental framework, each method is systematically assessed. Findings indicate that for designs with intricate counterexamples, BDD-based BMC excels in speed. However, in designs with straightforward counterexamples, while SAT-based BMC outperforms BDD-based BMC, explicit state-based BMC shows similar effectiveness.

The study, *Context-Bounded Model Checking of Concurrent Software* proposes a novel interprocedural static analysis based on model checking for detecting minor safety problems in unbounded parallelism concurrent programmes. The study is limited to runs with an arbitrary constant limiting the number of context transitions. Limiting the analysis to executions with a restricted number of context flips is unsound, but the analysis can still detect intricate defects and is sound up to the bound because a thread is fully probed for unbounded stack depth within each context. [21]. dfsd[18]

2.2.4 ESBMC

After exploring the detailed workings of static analysis, the capabilities of SMT solvers, and the effectiveness of bounded model checking, it's crucial to see how these elements function together in practical scenarios. A prominent example that embodies the integration of these principles is the Efficient SMT-based Bounded Model Checker, better known as ESBMC. This software is also the core of the project because the verification

function towards Linux Kernel is based on this software.
ESBMC is a context-bounded model checker, leveraging satisfiability modulo theories, designed for the verification of both single-threaded and multi-threaded C/C++ applications[22]. It can automatically validate both user-defined programme assertions and standard safety features (such as bounds check, pointer safety, and overflow). To access internal data structures and enable examination and extension at any point throughout the verification process, ESBMC provides C++ and Python APIs[23]. In addition, to more accurately support variables with finite bit width, bit-vector operations, arrays, structures, unions, and pointers, the team of ESBMC modified and expanded the encodings from earlier SMT-based bounded model checkers. The CVC3, Boolector, and Z3 solvers have been integrated with the CBMC front-end[24]. There are several important points about esbmc principles.ESBMC uses a k-induction algorithm to validate safety properties in C programs. A newly developed interval-invariant generator preprocesses the program, determining invariants from intervals and adding them as assumptions[25].

The k-induction method is a prominent technique in model checking. It is also the key that how ESBMC handles unbounded loops. This method uses temporal induction on finite-state machine time steps. Through an iterative process, it analyzes three distinct scenarios for every iteration defined by 'k'[26]. Firstly, the base case looks for potential counterexamples within k-loop unwindings. Secondly, the forward condition ensures that property P is upheld throughout k unwindings. Lastly, the inductive step ascertains the consistency of P's validity for subsequent unwinds. The safety property's compliance or violation is determined based on the verification conditions formulated for each unwinding.

ESBMC comprises several components that collaborate to ensure its robust operation. Figure 2.4 shows the architecture of the ESBMC tool and the detailed mechanism of how ESBMC verifies source files is fully discussed in Chapter 3.

2.2.5 Related works of Verification on Linux Kernel

In recent years, a variety of studies have been proposed for verification of the Linux kernel. Some of these researches take a distinct- give approach to solving vulnerabilities in different aspects. It is worth mentioning that the paper from[2],*'Linux kernel vulnerabilities: State-of-the-art defences and open problems'* also provides an overview of this topic. This study classifies 141 Linux kernel vulnerabilities published

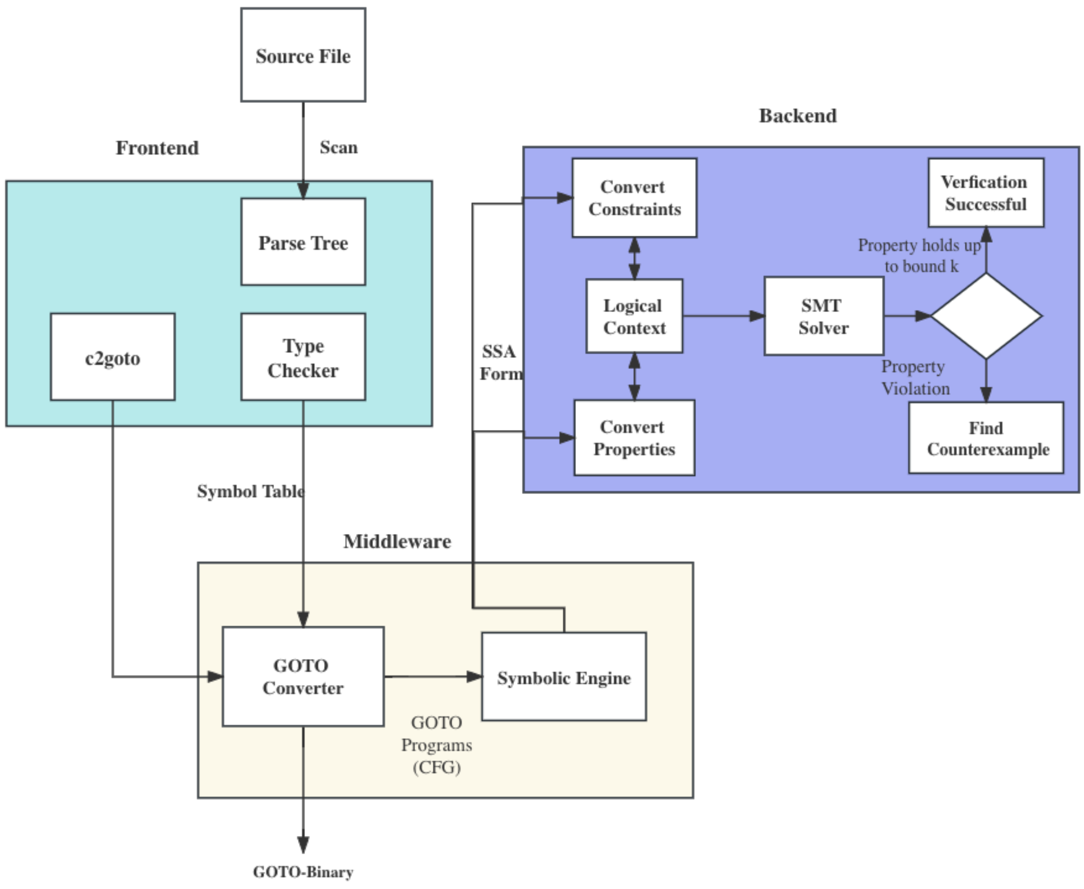

Figure 2.4: The architecture of ESBMC software.

in the CVE list between January 2010 and March 2011 and classifies the types of attacks that can exploit these vulnerabilities. It states that most vulnerabilities fall into 10 categories, e.g. Missing permission checks, Buffer Overflow, and Memory Mismanagement, based on the type of programming error developers make. This study also shows the distribution of vulnerabilities in the Linux kernel source tree. Some state-of-the-art prevention is introduced, but not deeply investigated.

The paper *Automatic Permission Check Analysis for Linux Kernel*[27] presents a static analysis framework for detecting permission check-related vulnerabilities in the Linux Kernel. This paper identifies the difficulty of finding omitted permission checks in the Linux kernel, which might result in security flaws. This paper introduces PeX, a static Permission check error detector for Linux, which is capable of scalably detecting any missing, inconsistent permission checks in the kernel codes. The technique involved is KIRIN, which is Kernel Interface Based In-direct Call Analysis. PeX automatically recognises all permission checks and infers the mappings between permission checks and privileged functions via the inter-procedural control flow graph created by KIRIN. Therefore, PEX checks all possible paths to compile to see the corresponding permission check before calling a privileged function.

In *'Model Checking Concurrent Linux Device Drivers'*," [28]. use predicate abstraction to propose DDVerify, a tool that automatically validates Linux device drivers. The authors take one step further on predicate abstraction, targeting and resolving the issues related to concurrency. This program reports benchmark tests based on Linux device drivers and also provides an accurate representation of the relevant kernel components. As concurrency plays a vital role in the execution of programs running in the kernel, this paper contributes the method to validate shared memory concurrent applications.DDVerify provides a concurrent model of the related parts of Kernel API. As a static verification tool, it produces an appropriate driver harness and inspects whether the driver violates the pre or postconditions of their kernel model.

The paper *'DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers'*[29] introduces DR. CHECKER, a comprehensive Linux kernel driver inspection tool, based on static analysis. The vast quantity of pointer code in these drivers poses some of the most challenging problems for static analysis. However, unlike the techniques used in ESBMC, which is model-bound checking, this tool uses flow-sensitive, context-sensitive, and field-sensitive analyses to track the flow of data through kernel drivers and identify potential security risks. These techniques allow the tool to accurately identify critical bugs in kernel drivers with high precision. In this project, the verification of Linux Kernel programs is achieved by modelling and abstraction on kernel operations based on ESBMC.

The paper *Analyzing and improving Linux kernel memory protection: a model checking approach* delivered a decent research on the security of Linux Kernel meomory. This study focuses on Linux kernel memory protection and methodically investigates for probable violations in Linux kernel architecture and implementation. The authors created a Murphi-based abstract model to discover numerous severe flaws in the present Linux kernel[30].The research confirmed the existence of these issues and created five Linux kernel patches, which are currently being merged into the mainline Linux kernel. According to the study, these patches entail relatively minor modifications to the existing code base and have low performance overhead. The authors validated the existence of these issues with the Linux kernel community and created kernel patches to address them. These fixes are backwards compatible with the existing Linux kernel code base and make only minor changes to the interfaces that manage kernel memory.

Chapter 3

Methodology and Implementation

This chapter aims to clearly state the challenge of the research question, and the approach taken to build the sound and reasonable deliverables. In section 3.1, the method, or approach this project used for modelling is addressed in detail. The section contains two subsections. The first subsection points out the gap, or challenge to verify Linux Kernel programs. The second subsection records how this difficulty is solved, and the intuitions and considerations behind the current approach. Section 3.2 focuses on the detailed implementation of the verification. This section records all modelling operations that are implemented and tested for verifying the vulnerabilities of kernels, and how these modelings are designed and built. This section also contains explorations of Linux Kernel source codes. Linux Kernel is an extremely complex and huge codebase, which consists of millions of lines of code. However, it is necessary to study the source codes in depth to understand how each module works and what the calling stack for a particular kernel function looks like. The modelling, the setting for pre-and-post conditions, and the verification progress are fully discussed in this section. Furthermore, some core codes of the implementation are shown with explanations.

3.1 Approach for Modeling

This section first addresses the challenge that exists while using ESBMC to verify the kernel programs and then elaborates on the approach taken by this project.

3.1.1 Gap

The Linux kernel, often cited as the heart of the Linux operating system, has millions of lines of code and contributions from thousands of developers worldwide. This means that maintaining the source codes and detecting possible defects becomes a demanding and challenging task. At the initial stage of the development of this project, parsing the kernel program is quite difficult. ESBMC, or other model checkers, do not need to compile and execute the program to detect the bugs. It is also necessary to know that ESBMC uses clang as the front end to parse the source codes. However, the Linux Kernel module requires compilation which relies on the Kbuild system. And Linux Kernel programs are usually compiled with GCC, instead of Clang. Even this project uses the Clang command *clang -Xclang -E -fsyntax-only*, which refers to parsing the codes to Abstract Syntax Tree(AST) only without compilation, the approach does not yield the expected results. Parsing kernel programs to AST still requires GCC-specific flags, that Clang can not deal with. The second challenge to successfully parse the kernel source file to the expected AST is related to the extremely deep and complex dependency of the Linux Kernel. After the exploration of source codes, it is found that the dependency in kernel source codes is complex.

As figure 3.1 shows, the source file that contains *kmalloc* function, *slab. h* is dependent on 5 header files: *linux/types.h*, *linux/overflow.h*, *linux/percpu_refcount.h* and *linux/GFP.h*. Furthermore, each header file also relies on several header files, resulting in deeper dependency. The deeper dependency is not shown in the figure since the recursive path of dependency is quite long. When the dependency goes deeper, some of the kernel source codes are dependent on more low-level assembly headers. When the Clang front parses the source file into an AST, it parses all the dependencies recursively to make sure the AST is complete and proper. This kind of nested dependency results in issues while parsing the source file. For example, the assembly-related header files could not be correctly handled. Even if the source file can be successfully parsed by Clang, processing it with ESBMC would still be immensely challenging due to its substantial size and complexity. Since converting the Linux Kernel program into the AST that ESBMC expects is infeasible at this stage, some necessary steps have to be taken to resolve this challenge. The next section explains what are these steps.

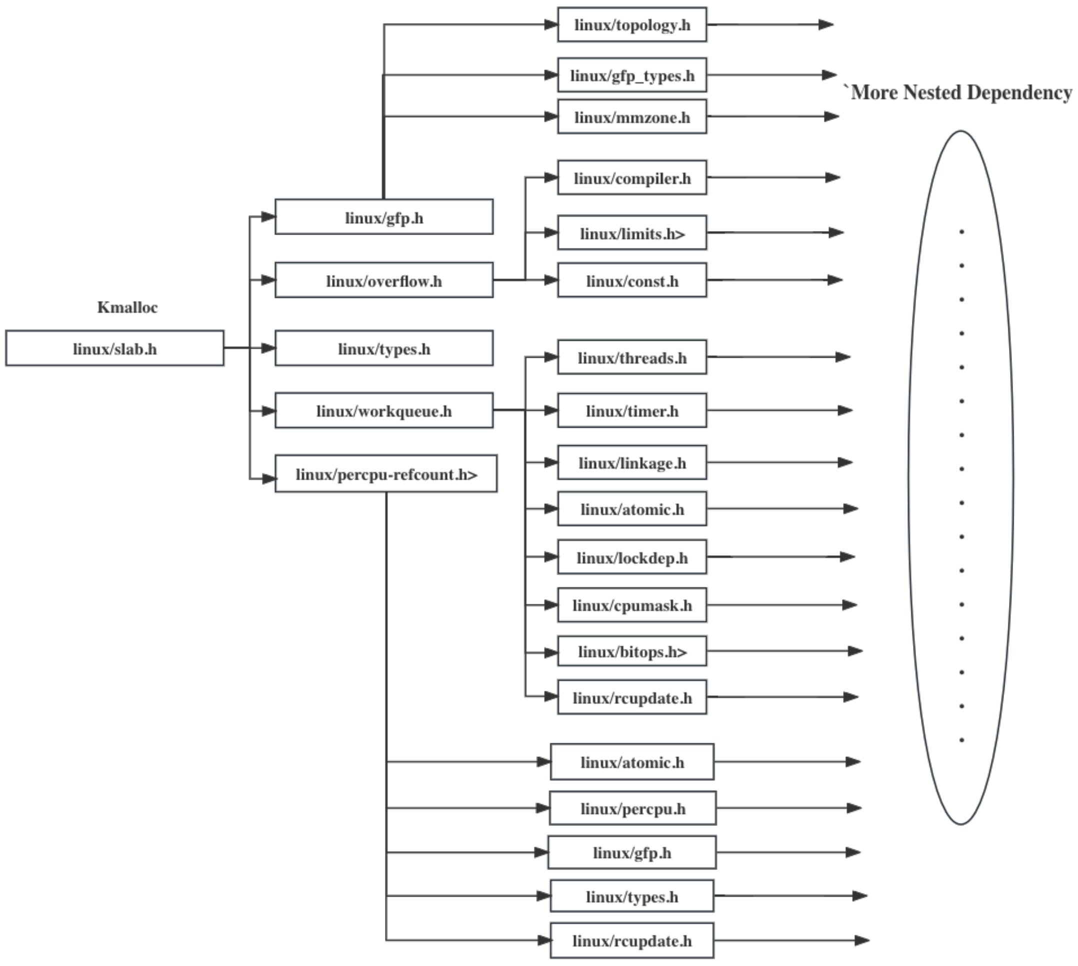

Figure 3.1: The recursive dependencies of slab. h, where the Kmalloc function is defined

3.1.2 Solutions

The previous sections list all gaps that exist in the verification of kernel programs. In this section, the approach taken by this project is discussed clearly. The first gap refers to the difference between Linux kernel programs and normal user-space C programs. Compiling Linux Kernel programs requires the participation of the Kbuild System, which relies on GCC. Although parsing the source file does not require the actual compilation, some required flags are GCC-specific, which can not be supported by Clang. One approach taken in this stage is to utilise the open-source tool to check what flags are involved while compiling and building the kernel. The open-source tool used here is *Bear*. Bear is a tool that creates a compilation database for clang tooling. The Clang project uses the JSON compilation database to offer details on how a single compilation unit is handled. With this, it is simple to start the compilation again with different programmes[31]. After this step, examining the output and information about the flags in the JSON file may help. The solution here is to remove those GCC flags and make sure the necessary clang flags are used in the command. In addition, the directory for parsing should be the correct one locally. However, the parsing encounters another issue though it is not complaining about the invalid flags anymore. While parsing the kernel source file, since the frontend recursively explores nested header files, several *asm* header files are parsed, which causes the problem. The Linux kernel makes extensive use of the inline assembly. While Clang supports inline assembly for many architectures, the specific dialect or the way inline assembly is used in the kernel might not always be compatible with Clang's expectations. In addition, Clang might not support or recognize all assembler directives used in kernel ASM headers, leading to parsing errors. As mentioned, the kernel uses a variety of GCC-specific macros and attributes that Clang doesn't recognize. It could result in errors especially if these macros are used in conjunction with assembly code. Thus, it is essential to come up with a solution that skips that deeply nested dependency to construct a simplified and clean AST for ESBMC to process.

The solution here is to mock the dependency for modelling the kernel functions. In this project, the solution to the gap addressed in the previous section is mocking the dependency, or more specifically, header files, of Linux Kernel to simplify the Abstract Syntax Tree(AST). Figure 3.1 shows the complex and deep dependency that is not expected for ESBMC to handle. This project uses the mocking method to cut down the deeply nested dependency, and only keep the necessary dependency for verification. The simplified header files are dependent on the kernel functions that the project aims to verify. As Figure 3.2 shows, the solution here cuts off all unnecessary header files by mocking the dependency. Mocking the dependency here refers to creating new kernel-specific libraries in *22goto/headers* directory in the ESBMC codebase. A directory named *ubuntu20.04/kernel_5.15.0-76* is created under *c2goto/headers* to mock the dependency of the kernel. By this approach, the parsing of kernel programs is executed based on the kernel library in ESBMC, instead of the actual kernel directory of the system. The Clang frontend first explores ESBMC libraries to inspect if the definition and the related header files of the kernel functions used in the source file can be found here. If the definitions are successfully located, then Clang is not going to explore the Linux Kernel source codes.

3.2 Modeling and Verification

After mocking the dependency to simplify the dependency, the project is dedicated to modelling some core kernel functions. Their core kernel functions play a vital role in memory management and concurrency. Any vulnerabilities may lead to unpredictable consequences. The method applied here was modelling the verified kernel operation by Abstraction. This method simplified and abstracted the core logic of kernel source codes. All modelling operations implemented will be fully discussed.

3.2.1 Modeling Dynamic Memory Allocation/destruction

This subsection focuses on the implementations of mocking dynamic memory allocation and destruction in kernel space. In user-space, *malloc()* is frequently used to apply for a dynamic memory space for storing data. In kernel-space, *kmalloc()* is in charge of the dynamic memory allocation. The function *malloc()*allocates memory on the heap, while kmalloc allocates memory from the kernel's primary memory area. This project models the core operations involved in kmalloc() and mocks its behaviour in user space to inspect the potential defects. The *kmalloc* function families are listed below[6].

Modeling Kmalloc(size_t size, gfp_t flags)

The project achieved modelling by abstracting the core behaviours from the complex source codes from kernel version *5.15.0-76*.The source codes of *kmalloc*in the Linux

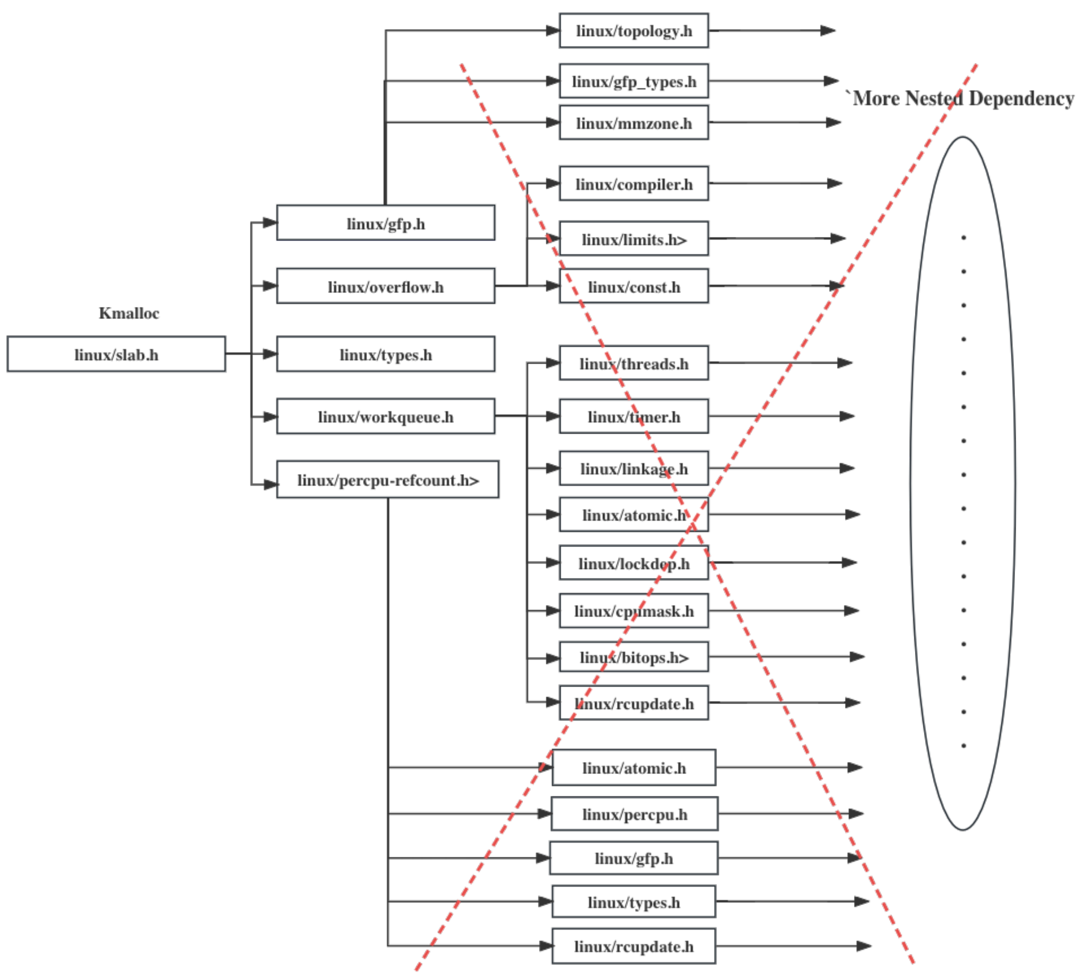

Figure 3.2: The mocking of the dependency simplifies the required header files to parse, and cut off unnecessary dependency

```
void *kmalloc(size_t size, gfp_t flags)
/**
* kzalloc - allocate memory. The memory is set to zero.
* @size: bytes of memory required.
* @flags: the type of memory to allocate.
*/
inline void *kzalloc(size_t size, gfp_t flags)
/**
* kmalloc_array - allocate memory for an array.
* @n: number of elements.
* @size: element size.
* @flags: the type of memory to allocate (see kmalloc).
*/
inline void *kmalloc_array(size_t n, size_t size, gfp_t flags)
/**
* kcalloc - allocate memory for an array.
* The memory is set to zero.
* @n: number of elements.
* @size: element size.
* @flags: the type of memory to allocate (see kmalloc).
*/
inline void *kcalloc(size_t n, size_t size, gfp_t flags)
   /**
    * krealloc - reallocate memory.
    * The contents will remain unchanged.
    * @p: object to reallocate memory for.
    * @new_size: bytes of memory are required.
```

Figure 3.3: The source codes about *kmalloc* in linux kernel 5.15[1]

```
static __always_inline void *kmalloc(size_t size, gfp_t flags)
{
        if (__builtin_constant_p(size)) {
#ifndef CONFIG_SLOB
                unsigned int index;
#endif
                if (size > KMALLOC_MAX_CACHE_SIZE)
                        return kmalloc_large(size, flags);
#ifndef CONFIG_SLOB
                index = kmalloc_index(size);

                if (!index)
                        return ZERO_SIZE_PTR;

                return kmem_cache_alloc_trace(
                                kmalloc_caches[
                    kmalloc_type(flags)][index],
                                        flags, size);
#endif
        }
        return __kmalloc(size, flags);
}
```

Figure 3.4: Another section of the source codes about *kmalloc* in linux kernel 5.15[1]

kernel are shown below. The codes are a bit complex. The function contains preprocessor directives like #ifndef CONFIG_SLOB. This means that the behaviour of this function can vary based on compile-time decisions, which can be influenced by configuration options. Such conditional compilation increases the complexity as you need to understand under which conditions specific code blocks get executed. However, based on the approach of this project, codes should only be parsed, instead of compiled and executed. It is infeasible to determine any expected behaviour in compile-time. Thus, this project first analysed the core behaviour of the function, which can be modelled through abstraction, and then decided the pre-and-post conditions that should be held for verification.

The first check is to see if the requested size is constant. If the size is constant and exceeds the maximum cache size, allocation is delegated to the *kmalloc_large* function.

Otherwise, for lesser sizes, it identifies the right memory cache index, validates it, and then allocates memory from the related cache, tracing the allocation process. To model this function, the objective of verification must also be confirmed. The objective decides what kind of attributes and operations are required when abstracting the original function. In this project, one assumption of vulnerability is that the improper values passed in as the parameters of *kmalloc* lead to unpredictable results. The assumption of vulnerability was tested. For example, the following codes passed negative size into the function.

```
char *buffer;
int size = -20; // Incorrect: Negative size value

printk(KERN_INFO "Loading example module...\n");

buffer = kmalloc(size, GFP_KERNEL);
if (!buffer)
    printk(KERN_ERR "Failed to allocate memory!\n");
else
{
    printk(KERN_INFO "Successfully allocated memory!\n");
    kfree(buffer);
}
```

Description of Vulnerability:

1.A negative value for allocation size In kernel, *size_t* is defined as *unsigned int*, which means it is expecting a non-negative value. However, when the user writes a kernel program, it is still possible that they use a negative value, e.g. -20 in the previous example, by mistake or by purpose. After testing, this situation leads to unpredictable behaviour, e.g. data corruption. This vulnerability exists since the negative parameter value passed into the function becomes an extremely huge value. For instance, if you try to assign a value of -1 to *size_t*, it will be stored as 4294967295 in decimal. Allocating such a huge size is definitely what the kernel is expecting.

2.Invalid GFP flags

The *gfp_t* flags in the Linux kernel refer to the set of flags used to modify the behaviour of memory allocation. The acronym gfp stands for "get free pages".The *gfp_t* refers to *unsigned int*. The risk here is that when the kernel program calls kmalloc with a gfp value that is negative, or not predefined in *gfp.h*. To solve this, the project implemented a function to check if the flag is valid .

kmalloc

The method of modeling *kmalloc* contains following steps:

1. The *kmalloc* function is a component inside Linux Kernel Memory Management. The source codes are located in *mm/slab.h*. As figure 3.1 shows, the header file *slab.h* is dependent on *linux/types.h*, *linux/overflow.h*, *linux/percpu_refcount.h* and *linux/gfp.h*. After deep analysis of what attributes were needed for verifying the potential vulnerabilities mentioned earlier, e.g. improper allocation size and invalid flags, it is confirmed that no further dependency on these 5 header files is necessary to the objective of the verification.

2. The *linux* directory is created under the kernel headers directory, which is created under *c2goto/headers*. These five header files, and the header file, *slab.h* where the verified function *kmalloc* is located, are created. In the header file *gfp.h*, different types of gfp flags are mocked from the Linux Kernel source codes. Those gfp flags are valid flags that can be used with memory allocation. The implementation of *gfp.h* is demonstrated below.

```
#define __GFP_DMA           0x01u
#define __GFP_HIGHMEM       0x02u
#define __GFP_DMA32         0x04u
#define __GFP_ZERO          0x40u
#define __GFP_NOWARN        0x80u
#define __GFP_REPEAT        0x400u
 ..........
#define __GFP_RECLAIM 0x01u
#define __GFP_IO      0x02u
#define __GFP_FS      0x04u
#define GFP_ATOMIC          (__GFP_HIGH | __GFP_ATOMIC | __GFP_KSWAPD_RE
#define GFP_KERNEL          (__GFP_RECLAIM  |  __GFP_IO  |  __GFP_FS)
#define GFP_KERNEL_ACCOUNT  (GFP_KERNEL  |  __GFP_ACCOUNT)
#define GFP_NOWAIT          (__GFP_KSWAPD_RECLAIM)
```

```
#define GFP_NOIO            (__GFP_RECLAIM)
.........
/* Convert GFP flags to their corresponding migrate type */
#define GFP_MOVABLE_MASK (__GFP_RECLAIMABLE|__GFP_MOVABLE)
#define GFP_MOVABLE_SHIFT 3
```

3. To verify the second vulnerability, a function inspecting the flags passed in is required. In this step, the project delivered a function to check if the gfp flag the user passed in refers to a valid flag defined in *gfp.h.* The implementation of this function is shown below.

```
static void check_gfp_flags(gfp_t flags)
{
  // Define all valid flags
  gfp_t valid_flags =
    __GFP_DMA | __GFP_HIGHMEM | __GFP_DMA32 | __GFP_ZERO |
    __GFP_NOWARN |__GFP_REPEAT | __GFP_NOFAIL |
     .....................
     | GFP_KERNEL_ACCOUNT | GFP_NOIO | GFP_NOFS |
    GFP_USER | GFP_DMA | GFP_DMA32 | GFP_HIGHUSER;

  // Check if any flag is set that is not in the list of valid flags
  assert((flags & ~valid_flags) == 0);
}
```

4. In this function, the flag value is passed in to check if it matches any value in the predefined flag value. The assertion is applied here to see if the property holds. The function is defined in *kernel.c*, a file that created to mock the kernel functions. The assertion implemented in this function checks if the post-condition holds after the verification. The postcondition here refers to the property that the flag used belongs to the set of predefined gfp flags in the kernel.

5.

6. The next step is to created *slab.h*, and declare kmalloc functions in this fille. The definition of kmalloc is implemented in *kernel.c*in *c2goto/library* directory. The *kmalloc* function in ESBMC was mocked by the original *kmalloc* function from

kernel source codes, with additional assertion to test the property. The first assertion in this function checks if the allocation size is greater than 0 and less than the *max_allocation_size*. This is the core operation to avoid illegal arguments. When a user writes a kernel program that involves kernel allocation with negative values, this assertion evaluates to false and fails the verification of *kmalloc* function, which achieves the objective of the modelling of *kmalloc*. Then if the verification of allocation size succeeds, the assertion on the value of the flag is executed. This assertion guarantees that any negative values of flags lead to failure of verification. This verification, together with the *check_valid_flags* function, prevents the unpredictable consequence caused by invalid flags.

7. The mocked *kmalloc* function involves *__kmalloc_large* when the size is greater than *KMALLOC_MAX_CACHE_SIZE*, and calls *__kmalloc* when the size does not exceeds the limit.

One important point to illustrate the mechanism of mocking is that this approach does not invoke the real kernel-space function. The user-space behaviours are designed and combined to simulate the kernel-space behaviour. In *__kmalloc* function, by simply calling *malloc*, which refers to memory allocation in user-space, the project simulates the behaviour of allocating memory in user-space.

kmalloc_array

The verification of *kmalloc_array* is similar to *kmalloc*. The signature of this function is *kmalloc_array(size_t n, size_t size, gfp_t flags)*. It has one additional parameter, size_t n, which refers to the number of allocated elements. This function involves three steps of verification. First, the assertion *assert(n ¿ 0)* verifies if the property of allocating a positive number of elements holds. After this assertion, the rest of the verifications are similar to the ones implemented in *kmalloc*. The assertions on allocating size and flags make sure that this function does not lead to any memory corruption.

kfree

The modelling of *kfree* is much more simple compared to that of *kmalloc*. There is no assertion here to verify if the pointer is null. This is because freeing a null pointer in C language is a safe operation. This means that if you pass a null pointer to free(), the function simply returns without doing anything.

3.2.2 Modeling Data Transmission Between Kernel and User Space

This subsection explains the method to model the data transmission operation between kernel and user space. It also demonstrates the implementation of the mocked kernel functions: *copy_to_user* and *copy_from_user*.
The Linux kernel operates in a privileged mode (kernel space), whereas user applications operate in a less-privileged mode (user space). For security and stability reasons, direct access between these two spaces is restricted. This is where copy_from_user() and copy_to_user() come into play. *copy_from_user* function transfers data from user space (where user applications operate) to kernel space (where the kernel runs). In contrast, *copy_to_user* function transfers data from kernel to user space.
Description of Vulnerability:
Null Pointer Dereferencing
Null pointer dereferencing happens when the program tries to access or modify a memory location corresponding to a null pointer, leading to undefined behaviour and typically resulting in a segmentation fault and program crash.
Memory Overflow with Large Data Transfers Memory overflow happens when data is written beyond the confines of allotted memory, frequently leading to buffer overflow attacks. This can cause data corruption, crashes, or unauthorised code execution by overwriting nearby memory areas. **Memory Boundaries Validation** Memory boundary violations occur when data is read from or written to memory regions that are outside the expected constraints. This can result in data corruption, unauthorised data access, or control flow hijacking.
copy_from_user
The method of modeling *copy_from_user* contains following steps:

1. copy_from_user() and its associated functions can be found in the architecture-specific portions of the kernel source, because the actual method of copying data may vary between different CPU architectures.For example, in the x86 architecture, the codes are located in *arch/x86/include/asm/uaccess.h*. To model this function, the project first created a *asm* directory under the *headers* directory in *c2goto* folder.

2. In the *uaccess.h* header files, several macros were defined. The declarations in this mocked file can be seen below. The *PAGE_SIZE* simulates the size of a page in Linux Kernel.*USER_SPACE_SPACE* is defined to simulate the user memory

```
// page size
#define PAGE_SIZE 4096
//only for simulate the user memory
#define USER_MEMORY_SPACE 10000
// only for simulate the kernel memory
#define KERNEL_MEMORY_SPACE 10000
//mock user memory
extern char user_memory[USER_MEMORY_SPACE];
//mock user memory
extern char kernel_memory[KERNEL_MEMORY_SPACE];
//simulate copy_to_user function in kernel space
unsigned long copy_to_user(void* to, void* from,
                            unsigned long size);
unsigned long copy_from_user(void* to, void* from,
                            unsigned long size);
```

3. The actual memory for user space and kernel space is architecture-specific and are only determined in runtime. Since the approach this project takes is detecting the errors without compiling and executing, a simulation that avoids acquiring the actual memory address is needed. This is why this project declares two arrays in *uaccess.h* to simulate the boundary of user space and kernel space. The arrays are only for the use of mocking the data transmission between different spaces. They were declared as global variables with *extern* keyword for external linkage. The simulated memory spaces were also applied for testing the behaviours of this kernel function in regression tests.

4. This project designs specific values for the size of simulated memory space.The *USER_MEMORY_SPACE* and *KERNEL_MEMORY_SPACE* are both defined with 10000. The simulated size is only for mocking the operations of data transmission kernel functions. It does not represent the actual memory spaces with any specific architecture.

5. The implementation *copy_from_user* is located in *c2goto/library/kernel.c*. In this implementation, there are several assertions to check the vulnerabilities. The following codes are parts of the implementation.

```
assert(to != NULL);
assert(from != NULL);
```

The first two assertions were applied to check the pointers. The first assertion is implemented to inspect if the destination pointer, which should be a kernel-space pointer, is null. It is illegal and unreasonable to copy the data from user space to a null pointer. This project has to guarantee that the data transmission takes place between valid pointers. The second assertion does the same inspection, but for the source pointer, which is in user space.

6. To prevent the defects, the inspections on the memory address were implemented.

```
assert((char *)to >= user_memory);
assert((char *)from >= kernel_memory);
memcpy(to, from, size);
```

This project applied assertions on the memory address of the source and destination pointers. It validates the user space pointer and ensures that the full range of memory being accessed is within the user space. If the user space pointer is faulty or the range extends beyond user space, the function does not perform the copy and normally returns the number of bytes that were not copied. This function transfers data from user space (where user applications operate) to kernel space (where the kernel runs). To simulate the copying operations, this mocked function calls C standard function *memcpy*.

copy_to_user

The method of modelling the *copy_to_user* is similar to the way of modelling *copy_from_user*. The only difference is that the source pointer in the function *copy_to_user* represents the memory address in user space, ad the destination pointer represents the memory address in kernel space. The implementation of this mocked function inspects the vulnerabilities while copying data from kernel space to user space.

3.2.3 Modeling Memory Leak using ESBMC

One of the ESBMC's verification scopes is memory leaks. When programming with low-level language, like C/C++, memory leaks are the problems that programmers have to be concerned about. Unlike high-level programming languages, e.g. Java, C/C++ does not support an automatic garbage collection mechanism. It means that manual memory management, e.g. freeing the memory blocks allocated after use, has to be done by programmers. Otherwise, memory leaks arise and lead to further risks for the programs. Since this project aims to achieve the verification in memory management in Kernel, .e.g *kmalloc* and *kfree*, the mechanism of how ESBMC handles the verification of memory leak is explained in this section. To understand ESBMC's method of dealing with malloc memory leaks, we must first grasp its goto-program and symbolic execution algorithms.

Goto-Program : Intermediate Representation

ESBMC and other CPROVER-based tools employ the goto-program as an intermediate representation (IR). At its core, the goto-program is a simple and language-independent representation of a program's logic that serves as an abstraction for analysis and manipulation. A programme is represented by the goto-program as a set of goto functions. Each of these functions contains a series of instructions that are linearized representations of the logic of the original programme. Assignments, assumptions, assertions, gotos, function calls, and other basic operations can be represented by the instructions. The goto-program has a big advantage in that it normalises the input source code. This means that in the goto-program, sophisticated programming constructions or syntactic sugar are reduced into simpler, standardised forms. This simplification makes analysis and transformations more consistent and simple.

The goto-programs directory's patterns *goto_convertt::do_mem*, *goto_convertt::do_malloc*, and *goto_convertt::do_alloca* explain how memory allocation actions in source code are transformed to an goto-program. The do_mem function is at the core of this transformation. The memory allocation function name is set (via the *func* variable) based on whether the operation is a malloc or an alloca. After retrieving the allocation type and size, the method prepares a sideeffect expression (new_expr). This *sideeffect* is critical since it effectively represents dynamic memory allocation. This expression is then added as an instruction to the goto-program, representing a memory allocation operation.

Symbolic Execution

Symbolic execution is an analysis technique that systematically investigates multiple execution routes of a program by substituting symbolic values for concrete input values. ESBMC uses symbolic execution to reason about all conceivable programme behaviours without having to run the programme on every possible input. Rather than using actual values, symbolic execution works on symbolic values, treating them as unknown variables. In symbolic execution, for example, instead of a variable x having a value of 5, x would be treated as an unknown value[32]. As the symbolic executor moves along a path, it acquires a set of constraints based on the criteria encountered. When these restrictions are satisfied, they offer input values that direct the programme down that particular execution path.

In ESBMC, the symbolic execution is presented through the *goto_symext* class functions like *goto_symext::symex_malloc*and *goto_symext::symex_mem*. This execution plays a pivotal role in abstractly simulating the program behaviour using symbolic values rather than concrete ones.
symex_malloc starts the memory allocation process by passing a flag indicating that it is a *malloc* operation to the symex_mem function. If the left-hand side (lhs) of a symex_mem operation is empty, the process is aborted. Otherwise, the code creates a symbolic representation of the memory that has been allocated. The dynamic object generation, denoted by the "dynamic_" prefix in the symbol name, is an important component of this symbolic execution method. These dynamic objects reflect the memory allocated by the *malloc* function symbolically. To accommodate for malloc's nondeterministic nature (it may fail and return a NULL pointer), the code includes an if statement (*if(!options.get_bool_option("force-malloc-success") && is_malloc)*) to mimic a nondeterministic choice between successful allocation and returning a NULL pointer.
Memory Leak Check
When the *free* function is invoked, ESBMC will perform a symbolic check to ensure that the pointer being freed points to a valid, previously allocated block of memory. If this is not the case, a dereference failure may occur. In addition, ESBMC will have a comprehensive trace of all memory allocations and deallocations at the end of the symbolic execution. By comparing them, ESBMC can determine whether any allocations had no matching deallocations. If such allocations exist, it indicates a memory leak, which ESBMC might disclose to the user.

3.2.4 Modeling Concurrency

Kernel-space operations frequently require the management of resources that can be used by numerous entities at the same time. A robust mechanism to prevent race conditions is necessary.

Spin Lock

Spin locks are synchronisation mechanisms used in kernel space to ensure that only one entity at a time has access to a shared resource. Mocking spin lock operations is typically advantageous for formally verifying kernel space code with ESBMC. By abstracting away the delicate complexities of concurrency, the verification process is simplified. Since some aspects of kernel-level concurrency might not be easily supported by ESBMC, this mock method helps ESBMC to detect potential vulnerabilities in a multi-thread environment. The declaration of this method locates in the same header directory as previous approaches. The header file is shown below.

```
/* SPDX-License-Identifier: GPL-2.0 */
#include <stdatomic.h>
#include <stdbool.h>
/* Declarations of mock spin lock behaviour */
/** mock spinlock struct */
#define SPIN_LIMIT 80
typedef struct {
    bool locked;
} spinlock_t;

void spin_lock_init(spinlock_t *lock);

bool spin_lock(spinlock_t *lock);

void spin_unlock(spinlock_t *lock);
```

In this header file, the project first defines a struct to model the spin lock used in the kernel. The struct *spinlock_t* contains a boolean value *locked* to indicate if the spin lock is acquired by any thread. If this value is set to false, then it means the lock is not acquired by any thread yet. If a thread acquires this lock, the boolean value should be modified to true. Three operations, initialization, lock, and unlock are declared. These operations are mocked from Linux Kernel, and use the argument with type *spinlock_t* to access the shared resource properly.

Description of Vulnerability:

Race Condition in Multi-threads Environment

When the kernel programs apply multi-threads for some particular operations that are related to shared resources, the modification and acquisition of the shared resource may lead to incorrect results. This pattern is caused by race conditions since some operations are not atomic. The context may switch to another thread in the middle of an operation. For example, when one thread reads the value of the shared resource, and then it plans to increment the shared resource with a specific value. However, the context switches right after it successfully read the value of the resource. Another thread may modify the shared resource. When the first thread finally get to increment the shared resource, it is modifying based on the previous value, which is inconsistent with the current value.

spin_lock_init

The modelling of this function is straightforward. This function aims to mock the initialization process of spin lock in the kernel space. The spin lock has state to indicate if the lock is acquired by any thread yet. The initialization in this implementation simply set the *locked* state to false. This boolean value indicates if the lock is acquired by any thread. It is set to false since the lock is not acquired by any threads in the initialization. The argument in this function is a pointer to the shared spin lock. However, it must be guaranteed that the lock should not be null. Thus this function first uses assertion to check if the lock is null. Otherwise, it initializes the spin lock by setting the value *locked* to false, indicating that no thread is holding this lock.

spin_lock

This implementation represents the core of the modelling of spin lock in concurrency. It is significant to make sure that the threads acquiring the lock behave in proper order. Any improper order of execution may lead to unexpected results. The implementation in this project first applies *__ESBMC_assert(lock != NULL, "The lock is null, verification failed");* to provide a more readable output for the users. Then this project designs a variable called retries to record the spin numbers of current threads. Unlike the mutex lock, the spin lock keeps spinning until it acquires the lock. The macro *spin_limit* is set to avoid the situation that the spin locks spins infinitely. Since ESBMC is based on the state exploring, the limitation on spin counts also prone the state space to a manageable size.

```
__ESBMC_atomic_begin();
if(lock->locked == false)
{
  lock->locked = true;
  __ESBMC_atomic_end();
  return true;
}
__ESBMC_atomic_end();
retries++;
```

In the spin lock, the most important point is that the operation has to be atomic to avoid thread switching. In ESBMC, *__ESBMC_atomic_begin* and *__ESBMC_atomic_end* guarantee that the operations between are considered atomic. In this function, the thread first checks if the lock is not acquired by any thread. If another thread is holding this lock, then it keeps spinning. Otherwise, it acquires the lock and set the *lock* value to true. The if statement and the modification on the boolean value are together considered as one atomic operation in this implementation. The function returns true if the thread successfully gets the spin lock. When the variable *retries* exceeds the limit of *SPIN_LIMIT*, it returns false. The operation *spin_unlock* simply checks if the current lock is null. If it is not, it changes the value of the *locked* inside of the struct *spinlock_t*.

Chapter 4

Evaluation

4.1 Description of Benchmark

The implementations of the verification methods in this project must be evaluated. The verification on Linux Kernel is challenging compared to that on normal C/C++ programs. Since ESBMC is a model checker that relies on bounded model checking, CBMC could be a possible benchmark suite to test and compare the performance with. However, CBMC is more well-known for user-space programs. This project designed its benchmark suite to evaluate the performance and correctness of the implementations. The benchmark suite designed was based on the generated tests. It aims to validate and ensure the correct functionality of core kernel operations, focusing on memory management, user-space interactions, and synchronization mechanisms.

4.2 Setup

The evaluations are set in a particular environment. The information on the experiment environment can be listed as follows:
CPU: Intel i7 9750H
Memory: 16 GB Operating System: Ubuntu 20.04
Clang version: Clang 11 ESBMC version: ESBMC version 7.3.0 64-bit x86_64 Linux

4.3 Objectives

The objective of the evaluations is to assess the capability of the implementations in this project to detect the known vulnerabilities in the kernel programs. The regression tests are designed to inspect if the results of the verification are as expected.

4.4 Results

This project creates 23 tests for the evaluation objective. The 23 tests cover the verification scope of memory management, data transmission and concurrency. In this section, the test results are evaluated. The expected outcomes and the efficiencies of the verification are also demonstrated in this section. The table below shows the results of the evaluation:

Test ID	Description	ESBMC flags	Expected	Actual
1	Copy kernel data from Kernel space	–unwind 200	Failed	Failed
2	Copy from user space	–unwind 200	Failed	Failed
3	copy the data from null pointer	–unwind 200	Failed	Failed
4	Copy data from user-space	–unwind 200	Successful	Successful
5	Spin Lock to prevent race condition	—-unwind 400 –context-bound 2	Failed	Failed
6	Spin Lock to prevent race condition	–unwind 400 –context-bound 2	Successful	Successful
7	Spin Lock to prevent race condition	–unwind 400 –context-bound 2	Successful	Successful
8	Copy data from Kernel space	–unwind 200	Failed	Failed
9	Copy to user-space	–unwind 200	Failed	Failed
10	Copy to user-space with null pointer	–unwind 200	Failed	Failed
11	Copy kernel data to user-space space	–unwind 200	Successful	Successful
12	invalid kmalloc allocation		Failed	Failed
13	kmalloc with invalid flags		Failed	Failed
14	invalid kmalloc flags		Failed	Failed
15	valid kmalloc allocation		Successful	Successful
16	valid kmalloc allocation		Successful	Successful

Table 4.1: Test Set for Evaluation Part 1

17	valid kmalloc flags		Successful	Successful
18	Copy from user-space to address in user-space	–unwind 200	Failed	Failed
19	copy from the data in user-space to user-space	–unwind 200	Failed	Failed
	Copy kernel data from Kernel space	–unwind 200	Failed	Failed
20	valid kmalloc allocation		Successful	Successful
21	invalid kmalloc allocation		Failed	Failed
22	valid kmalloc allocation with valid flags		Successful	Successful
23	invalid kmalloc flags		Failed	Failed

Table 4.2: Test Set for Evaluation Part 2

The results of the evaluation are shown. Each test case is labelled with the test ID, the description of the test, the expected verification result and the actual verification result. Each implementation of kernel verification in this project is tested in this evaluation, with multiple scenarios. Each test case has corresponding ESBMC flags to support the expected testing objective. These tests are created in the directory *regression/linux*. All 23 tests passed with the expected results.

4.4.1 Efficiency

The efficiency of these tests is also recorded. The test cases related to kernel allocation are test cases 12, 13, 14, 15, 16, 17, 20, 21, 22, 23. These test cases take about 0.29, 0.296, 0.307, 0.291, 0.290, 0.312, 0.296, 0.315, 0.30, and 0.293 seconds respectively. They only take a short time to test the functionality. The test cases related to data transmission are test cases 1, 2, 3, 4, 8, 9, 10, 11, 18, and 19. In contrast, these test cases take more time to evaluate. The time these test cases take is 6.81, 2.1, 1.5, 10.89, 6.71, 2.18, 1.3, 9.93, 6.94, and 6.71 seconds. The verification of these test cases takes more time due to the non-determining mechanism. There are three test cases generated for verifying spin lock. These test cases check if the shared resources are modified as expected in a multi-thread environment. It involves spinning and thread interleaving, which leads to a much longer verification time. The test cases 5, 6, and 7 take about 5.42, 70.288, and 8.1 seconds. The total time of these 23 test cases is 142.08 seconds.

4.4.2 Resolved Vulnerabilities

Based on the table, the verification of *kmalloc* and *kfree* succeeds in different contexts. The generated tests use several illegal and legal arguments for the size and flags. All tests come up with expected verification results. The verification of data transmission from kernel space to user space, or vice versa, achieves the expected results. The test cases that call*copy_from_user*, but use the kernel, or user address for both source and destination, fail in the process. This also represents the expected outcome since *copy_from_user* and *copy_to_user* are functions that support the data transmission between kernel and user space, instead of identical space. Lastly, the results of concurrency tests show the successful implementation to prevent the race condition at the kernel level.

4.4.3 Threats to Validity

The objective of the evaluation is achieved through the set of tests designed for this project. In addition, the robustness and effectiveness of the implementation can be inferred from the experimental results. Nonetheless, there are possible threats to the validity of the testing results. The first possible threat is that some vulnerabilities in test codes may not be detected through the approach of mocking the operations. For example, when verifying the data transmission in kernel and user space, the simulated memory spaces(for both kernel space and user space) may not represent the real-world memory space. The second threat is that the implementation of verifying concurrency problems in this project may take a very long time. As it can be seen from the efficiency information in the table, the verification on *spin_lock* involves exploring huge state spaces. Increasing the number of threads for testing may lead to a huge rise in the time spent on verification. This threat may affect the efficiency of the method in this project.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this project, we set out to improve the verification process for the Linux kernel by combining the ESBMC tool with the mocking technique. The major goal was to secure the dependability and correctness of essential kernel functions by imitating their behaviour through modelling, allowing us to submit them to rigorous testing and verification.

The project began by investigating the difficulties in verifying sophisticated software systems such as the Linux kernel. The inherent complexity of the kernel, combined with the possibility of security flaws and system crashes, highlights the necessity for rigorous verification procedures. Because of its formal verification features, ESBMC emerged as a strong candidate, allowing us to reason about the correctness of kernel functions under many scenarios. The implementation of this project completed the functionality of ESBMC, allowing this software to inspect Linux Kernel programs and find possible vulnerabilities properly.

The project designed a mechanism to integrate mocking into the verification workflow after a careful approach. It recreated these behaviours in controlled situations by building simulated implementations of critical kernel functions such as *kmalloc*, *kfree*, and *copy_from_user*. This allowed us to put these functions through a set of test cases and circumstances that would be impossible to reproduce in a real-world kernel context. The method achieves the verification of kernel codes in user space, which makes more easier to handle complex situations. The ability to isolate these routines and replicate their behaviour was critical in discovering corner cases, boundary issues, and potential memory leaks that would have gone undiscovered otherwise.

Although concurrency is a complex mechanism to handle especially in kernel space, this project implemented the method to model the concurrency in kernel space and mock the *spin_lock* related functions. It modelled the behaviors of spinning and correctly acquiring the lock and ultimately prevented the race condition. Several test cases were created to evaluate the functionality and efficiency of the implementations in this project. The experimental results indicated that the method designed achieved the objectives of this project.

5.1.1 Limitations and Future Work

While this project represents a solid step forward in improving the Linux kernel verification process, there are various options for future investigation and improvement:

Enhanced Mocking Techniques: Mocking's usefulness is dependent on the accuracy of the mock implementations. More advanced strategies for developing mimic behaviours that correctly mirror real-world events could produce even more accurate outcomes.

Integration with Real-World Kernels: While mocking provides a controlled environment for testing, it is critical to validate the technique's effectiveness on real-world kernels. It will be beneficial to integrate the mocking approach with actual kernel builds and evaluate its impact on verification accuracy.

More comprehensive benchmark suite: Another limitation of this project is that the benchmark suite could be more comprehensive and cover more aspects of testing. It could happen that the benchmark suite used in this project can not fully evaluate the implementation in every aspect.

Integration with Other Verification Tools: ESBMC is simply one of many accessible verification tools. Investigating the possibility of combining the mocking technique with additional formal verification tools could result in a more thorough verification strategy.

In summary, this project successfully delivered a feasible solution to detect the vulnerabilities in Linux Kernel programs. It proposes an alternative approach to enhancing the Linux kernel verification process through the use of ESBMC and mocking techniques. We exposed essential kernel functions to intensive testing and analysis by establishing simulated environments for them, revealing vulnerabilities and concerns that might threaten system stability and security. Mocking's successful integration into the verification workflow demonstrates its potential to improve the reliability of complex software systems.

Bibliography

[1] Linux Kernel Development Team. Specific component of linux kernel source code, 2021. Accessed: 2023-07-01.

[2] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. *Proceedings of the 2nd Asia-Pacific Workshop on Systems, APSys'11*, 07 2011.

[3] Mikhail R. Gadelha, Rafael Menezes, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis Nicole. Esbmc: Scalable and precise test generation based on the floating-point theory. In Heike Wehrheim and Jordi Cabot, editors, *Fundamental Approaches to Software Engineering*, pages 525–529, Cham, 2020. Springer International Publishing.

[4] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel.* Oreilly amp; Associates Inc, 2005.

[5] Robert Love. *Linux Kernel Development.* Addison-Wesley Professional, 3rd edition, 2010.

[6] C.R. Maruthi and R. Bharadwaj. *Mastering Linux Kernel Development: A Kernel Developer's Reference Manual.* Packt Publishing, 2017.

[7] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, 2004.

[8] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[9] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[10] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[11] Peter T Breuer and Marisol Garciá Valls. Static deadlock detection in the linux kernel. In *International Conference on Reliable Software Technologies*, pages 52–64. Springer, 2004.

[12] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Marco Bozzano, Alessandro Cimatti, Gabriele Colombini, Veselin Kirov, Roberto Sebastiani, et al. The mathsat solver–a progress report. *Proc. Workhop on Pragmatics of Decision Procedures in Automated Reasoning 2004 (PDPAR 2004)*, 2004.

[15] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*, pages 174–177. Springer, 2009.

[16] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[17] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[18] Nina Amla, Robert Kurshan, Kenneth L McMillan, and Ricardo Medel. Experimental analysis of different techniques for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 9*, pages 34–48. Springer, 2003.

[19] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19:7–34, 2001.

[20] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 85–96. Springer, 2004.

[21] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 93–107. Springer, 2005.

[22] Esbmc. `https://www.esbmc.org`.

[23] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. Esbmc 5.0: an industrial-strength c model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 888–891, 2018.

[24] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2011.

[25] Mikhail R Gadelha, Felipe Monteiro, Lucas Cordeiro, and Denis Nicole. Esbmc v6. 0: Verifying c programs using k-induction and invariant inference: (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25*, pages 209–213. Springer, 2019.

[26] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. Handling unbounded loops with esbmc 1.20. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 619–622, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] Jinmeng Zhou, Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed Azab, Ruowen Wang, Peng Ning, and Kui Ren. Automatic permission check analysis for linux kernel. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[28] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent linux device drivers. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 501–504, 2007.

[29] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, 2017.

[30] Siarhei Liakh, Michael Grace, and Xuxian Jiang. Analyzing and improving linux kernel memory protection: a model checking approach. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 271–280, 2010.

[31] Bear. `https://github.com/rizsotto/Bear`.

[32] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.