Contributions for Improving Debugging of Kernel-level Services in a Monolithic Operating System

Tegawendé F. Bissyandé

N° d'ordre: 4768

THÈSE

Présentée devant

L'UNIVERSITÉ DE BORDEAUX
École Doctorale de Mathématiques et Informatique

pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX
Mention INFORMATIQUE

par

Tegawendé François d'Assise BISSYANDÉ

Équipe d'accueil : PROGRESS
École Doctorale : Mathématiques et Informatique
Composante universitaire : LABRI

Titre de la thèse :

Contributions for Improving Debugging of Kernel-level Services in a Monolithic Operating System

Contributions à l'Amélioration du Débogage des Services Noyau dans un Système d'Exploitation Monolithique

Soutenue le 12 Mars 2013 devant la commission d'examen

M. :	Xavier	BLANC	Président
MM. :	Yves	LEDRU	Rapporteurs
	Olaf	SPINCZYK	
MM. :	Julia	LAWALL	Examinateurs
	David	LO	
	Laurent	RÉVEILLÈRE	
M. :	Gilles	MULLER	Invité

à Josy pour avoir été là de bout en bout,
à tonton Michel pour le soutien "paternel",
et à ma maman Lucie pour son courage et son abnégation.

Acknowledgements

This dissertation would not have been possible without the support, help or guidance of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of my university studies. Among those individuals are Dr. Michel M.D. Nikiéma (*tonton Michel*) who stood as a father to me. I also acknowledge the courage of my mother, Lucie Bissyandé, who, in uncertain times, did not stop fighting for our interests. Ouindpouiré Josiane Sedogo supported me throughout this PhD experience and I shall remain grateful to her. I would also like to mention my brothers and sister, especially Flore Bissyandé who was always supportive.

In the research community, I am grateful to several researchers and teachers who guided me towards the realization of this dissertation. I am specially thankful for meeting Olivier Beaumont, Director of Research at INRIA, who was the best teacher I got in engineering school.

I would like to express the deepest appreciation to Professor Xavier Blanc, Professor at the University of Bordeaux, who gracefully accepted to chair my defense committee.

Second, a special thank to Professors Yves Ledru from the University of Grenoble, and Olaf Spinczyk from the University of Berlin, who accepted to review my thesis document before the committee can sit. I would also like to thank other committee members, Julia L. Lawall, Director of Research at INRIA, David Lo, Assistant Professor at the Singapore Management University, and Laurent Réveillère, my thesis supervisor, Associate Professor at the Bordeaux Institute of Technology. My gratitude goes also to Gilles Muller, Director of research at INRIA whose guidance is embedded in the entire dissertation.

Next, I would like to acknowledge the friends I have made here in France:

- friends in CVT at LaBRI: Rémi Laplace (and JDB), Jonathan Ouoba, Jérémie Albert, Damien Dubernet, Hugo Balacey, Daouda Ahmat, Cyril Cassagnes, Jigar Solanki, Vincent Autefage, Télesphore Tiendrebeogo and Sebastien Bindel.

- fellow PhD students at LaBRI: Thomas Morsellino, Florent Foucaud, Vincent Filou, Vincent Rabeux and Yi Ren.

Finally, I am as ever indebted to God, who made all things possible.

Abstract

Despite the existence of an overwhelming amount of research on the quality of system software, Operating Systems are still plagued with reliability issues mainly caused by defects in kernel-level services such as device drivers and file systems. Studies have indeed shown that each release of the Linux kernel contains between 600 and 700 faults, and that the propensity of device drivers to contain errors is up to seven times higher than any other part of the kernel. These numbers suggest that kernel-level service code is not sufficiently tested and that many faults remain unnoticed or are hard to fix by non-expert programmers who account for the majority of service developers.

This thesis proposes a new approach to the debugging and testing of kernel-level services focused on the interaction between the services and the core kernel. The approach tackles the issue of safety holes in the implementation of kernel API functions. For Linux, we have instantiated the Diagnosys automated approach which relies on static analysis of kernel code to identify, categorize and expose the different safety holes of API functions which can turn into runtime faults when the functions are used in service code by developers with limited knowledge on the intricacies of kernel code.

To illustrate our approach, we have implemented Diagnosys for Linux 2.6.32 and shown its benefits in supporting developers in their testing and debugging tasks. The contributions of this thesis are:

- We identify the interface of kernel exported functions as a sweet spot at which it is possible to interpose the generation of debugging information, in a way that improves debuggability but does not introduce an excessive runtime overhead.
- We identify safety holes as a significant problem in the interface between a service and the kernel. Indeed, of the 703 Linux 2.6 commits for which the changelog refers explicitly to a function exported in Linux 2.6.32, 38% corrected faults that are related to one of our identified safety holes. Thus, although we may assume that in-tree kernel code is much more thoroughly tested than new service code under development, violations of these safety holes have still caused numerous crashes and hangs. In this thesis, we propose an extended characterization of these safety holes for the Linux kernel.
- We propose an approach to allow a service developer to seamlessly generate, integrate, and exploit a kernel debugging interface specialized to the service code. This approach has a low learning curve, and in particular does not require any particular Linux kernel expertise.
- Using fault-injection experiments on 10 Linux kernel services, we demonstrate the improvement in debuggability provided by our approach. We find that in 90% of the cases in which a crash occurs, the log contains information relevant to the origin of the defect, and in 95% of these cases, a message relevant to the crash is the last piece of logged information. We also find that in 93% of the cases in which a crash or hang occurs, the log information reduces the number of files that have to be consulted to find the cause of the bug.
- We show that the generated debugging interface incurs only a minimal runtime overhead on service execution, allowing it to be used up through early deployment.

Beyond operating systems software, the Diagnosys approach described in this thesis can be applied to any software based on the plug-in model, where extension code is written to interact and complement a core software through an ever-expanding interface. The proposed solution thus opens up new possibilities for improving the debugging of such software.

Key words

Diagnosys, Debugging, Wrappers, Linux, Device Drivers, Software Engineering, Reliability, Testing.

Résumé

Alors que la recherche sur la qualité du code des systèmes a connu un formidable engouement, les systèmes d'exploitation sont encore aux prises avec des problèmes de fiabilité notamment dûs aux bogues de programmation au niveau des services noyaux tels que les pilotes de périphériques et l'implémentation des systèmes de fichiers. Des études ont en effet montré que que chaque version du noyau Linux contient entre 600 et 700 fautes, et que la propension des pilotes de périphériques à contenir des erreurs est jusqu'à sept fois plus élevée que toute autre partie du noyau. Ces chiffres suggèrent que le code des services noyau n'est pas suffisamment testé et que de nombreux défauts passent inaperçus ou sont difficiles à réparer par des programmeurs non-experts, ces derniers formant pourtant la majorité des développeurs de services.

Cette thèse propose une nouvelle approche pour le débogage et le test des services noyau. Notre approche est focalisée sur l'interaction entre les services noyau et le noyau central en abordant la question des "trous de sûreté" dans le code de définition des fonctions de l'API du noyau. Dans le contexte du noyau Linux, nous avons mis en place une approche automatique, dénommée Diagnosys, qui repose sur l'analyse statique du code du noyau afin d'identifier, classer et exposer les différents trous de sûreté de l'API qui pourraient donner lieu à des fautes d'exécution lorsque les fonctions sont utilisées dans du code de service écrit par des développeurs ayant une connaissance limitée des subtilités du noyau.

Pour illustrer notre approche, nous avons implémenté Diagnosys pour la version 2.6.32 du noyau Linux. Nous avons montré ses avantages à soutenir les développeurs dans leurs activités de tests et de débogage. Les contributions de cette thèse sont les suivantes:

- Nous identifions l'interface des fonctions exportées du noyau comme un endroit opportun où il est possible d'interposer la génération des informations de débogage. Cette interposition est réalisée de façon à améliorer le débogage sans introduire un surcoût d'exécution excessif.

- Nous identifions les trous de sécurité comme un problème important dans l'interface entre les service noyau et le noyau central. En effet, parmi les 703 commits de Linux 2.6 résolvant un bogue lié explicitement à l'usage d'une fonction d'API, 38% ont corrigé des fautes liées à un des trous de sûreté que nous avons identifiés. Ainsi, bien que nous puissions supposer que le code dans le noyau maintenu par les développeurs de Linux est bien mieux testé qu'un nouveau service en développement, les violations des trous de sécurité y ont causé de nombreux crash et gels (*hangs*). Dans cette thèse, nous proposons une charactérisation étendue de ces trous de sûreté pour le noyau Linux.

- Nous proposons une approche permettant, de façon transparente, à un développeur de services de générer, d'intégrer et d'exploiter une interface de débogage spécialisée au code d'un service. Cette approche a une faible courbe d'apprentissage et, surtout, ne nécessite aucune expertise particulière du noyau Linux.

- En se basant sur des expériences d'injections de fautes sur 10 services noyaux, nous avons démontré l'amélioration que notre approche apporte en terme de débogage. Nous constatons que dans 90% des cas dans lesquels un crash se produit, le journal de debogage contient des informations relatives à l'origine de la faute, et que dans 95% de ces cas, un message pertinent au crash est la dernière information consignée. Nous avons également constaté que dans 93% des cas où un crash se produit, ou où le système est gelé, les informations présentes dans le journal permettent de réduire le nombre de fichiers à visiter en phase de debogage afin de trouver la cause du bogue.

- Nous montrons que l'interface de débogage générée rajoute une surcharge d'exécution minimale sur l'exécution normale de service, lui permettant d'être utilisée jusqu'en phase de prédéploiement.

Au-delà du code de systèmes d'exploitation, l'approche Diagnosys décrite dans cette thèse peut être appliquée à toute application basée sur le modèle de plug-in, où du code d'extension est produit pour interagir et compléter le code de base d'une application grâce à une interface en constante expansion. La solution proposée ouvre ainsi de nouvelles possibilités pour l'amélioration de la mise au point de tels applications.

Mots clés

Diagnosys, Débogage, Linux, Pilotes de périphériques, Génie Logiciel, Fiabilité, Test

Aperçu du document

Nous proposons un résumé en français de chacune des 3 parties et des chapitres qui composent ce document de thèse.

Chapitre 1. En Introduction, nous motivons la thèse dans le contexte des systèmes d'exploitation (SE) *monolithiques*, les plus utilisés dans nos environnements quotidiens. Ces types de SE ne séparent pas le coeur du noyau des extensions, comme les pilotes de périphériques, qui sont souvent réalisées par des tiers. Conséquemment, une faute introduite dans une de ces extensions peut entrainer la défaillance de tout le système. Malheureusement, l'utilisation d'outils de détection de bugs ou la mise en place d'un processus systématique de tests sont difficiles. Nous proposons donc une approche basée sur l'identification de "trous de sûreté" (*safety holes*) dans les interfaces de programmation (APIs) du noyau, et l'utilisation de techniques d'analyse statique pour les localiser. La réalisation de l'outil Diagnosys a servi pour automatiser la détection des trous et la construction des interfaces de débogage tout en facilitant les tests. Nous avons validé ces contributions de façon expérimentale.

Partie I: Contexte et Etat de l'Art

Cette partie revient sur le contexte de la thèse avec un chapitre aprofondissant les motivations.

Chapitre 2. Nous nous concentrons sur Linux, un SE monolithique. Nous décrivons les observations de notre étude des APIs de noyau de Linux. Ces APIs reposent sur des préconditions qui sont rarement controlées lors des appels. Malheureusement, elles sont peu documentées et évoluent constamment. De telles propriétés favorisent (1) la mauvaise utilisation des APIs par des développeurs ne pouvant les maîtriser suffisamment, et (2) l'apparition de défauts dans les services noyau utilisant ces APIs.

Chapitre 3. Dans ce chapitre, nous nous intéressons aux types des fautes les plus fréquentes dans le noyau de Linux. La catégorisation des fautes est empruntée à une étude de Chou et. [CYC+01]. Nous discutons également les types de défaillances que ces fautes entraînent (crashes et gels). Enfin, nous passons en revue diverses techniques qui permettent soit de détecter les fautes, soit d'en atténuer les conséquences. Toutefois, nous constatons que la portée de ces techniques est souvent trop limitée.

Chapitre 4. De nombreux facteurs fragilisent l'API de Linux. Entre autres, il y'a :

- le manque de robustesse, voulu pour ne pas pénaliser les performances
- la constante évolution des fonctions proposées et leurs signatures
- le trop grand nombre de fonctions et l'absence de documentation pertinente

De plus, le débogage est difficile car les messages d'information fournies dans le journal de crash sont souvent incomplets. L'analyse statique pour détecter en amont des bogues renvoie trop de faux positifs, et l'analyse dynamique pénalise les performances. Finalement, l'analyse statique post-mortem des messages d'erreur en cas de crash n'est pas satisfaisante car ces messages ne contiennent pas toujours les informations pertinentes. Tous ces éléments plaident en faveur de nouveaux outils automatisés d'aide à l'analyse et à la mise au point des services noyau tels que les pilotes de périphériques.

Chapitre 5. Dans nos travaux, nous avons priviligié l'analyse parmi les diverses techniques de recherche de bogues. Dans ce chapitre, nous rappelons des notions importantes en analyse statique : complétude, correction, faux positifs, faux négatifs. Nous présentons ensuite Coccinelle [PLHM08], l'outil que nous utilisons pour programmer nos analyses, ansi que ses "*semantic matches*" qui permettent d'identifier un morceau de code dont l'un ou tous les flux de contrôle vérifie certaines conditions. Nous montrons ensuite comment nous Coccinelle, bien qu'étant initialement conçu pour uniquement des analyses intraprocédurales, peut être utilisé pour des analyses interprocédurales.

Chapitre 6. Ce chapitre rappelle les points importants de la partie I et annonce les contributions décrites dans la deuxième partie.

Partie II: Identification des Trous de Sûreté

Chapitre 7. Ce chapitre revient en détail sur la définition de la notion de trou de sûreté (*safety hole*). Il s'agit d'un fragment de code du noyau qui peut mener à une défaillance si sa précondition implicite n'est pas respectée. Des exemples réels de trous de sûretés tirés du noyau de Linux sont discutés dans ce chapitre. Nous faisons ensuite la distinction entre un *entry* safety hole, où le noyau est appelé erronément par le code du pilote, et un *exit* safety hole où le résultat d'un appel au noyau est mal utilisé par le pilote. Une taxonomie des trous de sûreté est ensuite proposée: pour chaque type de bogue défini dans l'étude empirique de Palix et al. [PST+11], nous définissons une paire d'entry et exit safety holes.

Chapitre 8. Dans ce chapitre nous présentons les différents outils réalisés pour mettre au point l'approche Diagnosys. Il s'agit de:

- *SHAna*, l'analyseur statique qui parcours le noyau à la recherche des trous de sûreté et qui propose des préconditions d'utilisation en rapport avec les défauts potentiels.
- *DIGen*, le générateur de l'interface de débogage qui produit une enveloppe pour chacune des fonctions de l'API ayant au moins un trou de sûreté découvert par SHAna.
- *CRELSys*, le mécanisme de sauvegarde persistant des logs et de redémarrage chaud.

Chapitre 9. Ce chapitre illustre l'utilisation de Diagnosys sur deux défauts connus, menant dans le premier cas à un crash du système et dans le second à son gel. Nous montrons que le dernier message enregistré par Diagnosys dans le buffer circulaire de CRELSys est le très fréquemment pertinent pour identifier le défaut ayant mené à la défaillance du système.

Partie III: Evaluation de l'Impact de l'Approche

Chapitre 10. Dans ce chapitre nous évaluons le besoin et les bénéfices de l'approche Diagnosys. Ainsi, nous montrons d'abord que SHAna permet de découvrir des milliers de trous des sûreté. Ensuite, nous montrons, grâce à une étude des mises à jour (commits) du noyau que 38% des corrections sur l'utilisation des fonctions d'API étaient liées à des trous de sûretés. Enfin grâce à une expérimentation ciblée durant laquelle nous avons injecté des fautes dans 10 services noyau pour produire des crashes et des gels, nous montrons que Diagnosys est efficace pour fournir des informations pertinentes sur la cause et l'origine d'un défaut.

Chapitre 11. Ce chapitres est consacré à l'évaluation du coût supplément de notre approche Diagnosys. D'une part, nous évaluons le coût de certification des résultats de SHAna pour détecter les faux positifs. D'autre part nous évaluons le la perte de performance à l'exécution liée à la vérification des préconditions. Ces évaluations conduisent à admettre que le coût de Diagnosys est supportable.

Chapitre 12. Le douxième chapitre conclue cette thèse. Nous y rappelons les contributions de nos travaux avant d'esquisser les perspectives. D'une part, nous discutions comment notre démarche adoptée peut être appliquée à d'autres applications à base de plugins. D'autre part, nous émettons l'idée de la correction automatique des bogues découverts avec Diagnosys.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"UNIX is basically a simple operating system,
but you have to be a genius to understand the simplicity."
Dennis Ritchie

Contents

Modern computers have changed many aspects of today's world. They have now improved almost every facet of our lives, by supporting medical procedures, enhancing bank transactions, delivering more reliable and faster communication, increasing the efficiency of automobiles, and even basic home appliances such as refrigerators. Though the hardware of such computers come in different sizes, a unique software is often dedicated to control all operations, direct the input and output of data, keep track of files and control the processing of other computer programs: it is the Operating System (OS), which serves as an interface between the computer and the user. Its role is to manage the functioning of computer hardware, run application programs and share resources among computing tasks by allocating CPU time and memory.

Unfortunately, computers crash. In the early 90s, Gray has performed a survey which established that the main cause of outage had shifted from hardware and maintenance failures to failures in software [Gra90]. Software failures have now become the dominant cause of system unavailability. OS failures are particularly serious as they suppose a failure of the kernel to continue its services and thus, inevitably, lead to down time.

In common monolithic OS architectures, this reliability issue is exacerbated all kernel-level code, running in privileged mode, i.e., kernel mode, have direct access to all hardware and memory in the

system. To cope with this issue, alternate micro-kernel-based architectures have been proposed, where only the near minimum of software required to implement an OS is kept to be run in kernel mode. The remainder in the collection of software constituting an OS is run in user mode where failures would have a lesser impact on the entire system. Nonetheless, despite the long time-established performance of micro-kernels [HHL+97], monolithic architectures are still widespread in commodity OSes where users increasingly require more reliability and safety.

A monolithic OS is continually enriched with new kernel-level services, such as device drivers or file systems, for implementing new functionalities or supporting new devices. When the implementation of such software is faulty, it compromises the reliability of the entire OS. A variety of approaches have been developed to improve the process of developing driver code, including Devil [MRC+00], Dingo [RCKH09], and Termite [RCK+09], which propose to derive driver implementations from specifications, and RevNic [CC10], which proposes to derive new drivers from existing drivers for other OSes. Other approaches such as Nooks [SBL03], SafeDrive [ZCA+06], TwinDrivers [MSZ09] and DD/OS [LUSG04] try to protect the kernel from driver defects.

Nevertheless, because of the heavyweight nature of these solutions and their often narrow scope, industry practice is still to write drivers by hand, and to run them at the kernel level, with full privileges. In this context, developers of kernel-level services rely on classic software analysis tools and debugging techniques. Advances in bug-finding tools [ECCH00, LBP+09, LZ05, BBC+06] and specialized testing techniques [MC11, KCC10] have eased but not yet fully solved the challenges in producing more reliable kernel-level services. In Linux, studies have shown that such services, particularly device drivers, contain up to 7 times more bugs than other parts of the kernel [CYC+01, PST+11]. Palix *et al.* have furthermore established that each release of the Linux kernel contains between 600 and 700 bugs [PST+11], suggesting that :

- Extensive usage of existing bug-finding tools is tedious. Indeed, existing tools are known to produce significant numbers of false positives which can deter developers [LBP+09].

- Testing is not fully thorough in early-stages of driver development. Testing tasks can be cumbersome in the case of OS services as they often require frequent reboots and special environments to recover and analyses the outputs of test scenarios.

1.1 This thesis

The thesis presented in this document proposes an alternate approach to support developers in the testing and debugging of kernel-level services in a monolithic OS. The thesis tackles the development difficulties of kernel-level services by focusing on the issues that appear in the interaction between a service and the rest of the kernel. We propose an approach for monitoring this interaction to accurately inform the developer of any relevant problem that arises during testing.

Experimental work in the course of producing the thesis was done with the Linux kernel, a monolithic OS architecture that has progressively gained popularity. Linux has indeed expanded to an increasing number and range of platforms, from embedded systems to supercomputers. The proposed approach, however, is relevant to any monolithic OS architecture.

We propose the Diagnosys [BRLM12] approach which starts with an analysis of the kernel programming interface to identify and expose safety holes that may lead to faults when a kernel-level service misbehaves in its interaction with other kernel code. The tool that we have developed for the Linux kernel creates debugging interfaces that are specifically tailored to each kernel-level service

under development, and that allow to record in a crash-resilient log all the flagged misuses of kernel functions.

1.2 Contributions

The contributions of this thesis are multiple. We first study kernel code and identify safety holes in kernel APIs as a probable source of difficulties. We then present the steps that we have taken to systematically categorize safety holes in the kernel and how to detect them using static analysis. We also discuss the implementation of Diagnosys, in which, information on safety holes is leveraged to construct debugging interfaces. This implementation also offers a lightweight reboot system, built on top of existing kernel dumping solutions, to substantially ease testing. Finally, we evaluate our approach by discussing its benefits and validating its practical usefulness and usability.

Characterization of safety holes. We succinctly discuss the notion of *safety hole* and its characterization in the kernel programming interface. We highlight how safety holes pose a serious problem to kernel developers. We also provide an extensive exploration of the static analysis rules that we have written to identify instances of safety holes in the Linux kernel.

Diagnosys. We present Diagnosys, an automated tool for identifying safety holes and monitoring the execution of relevant code to improve testing tasks in the early stages of kernel-level service development. We describe in details the implementation of Diagnosys, including the static analysis, the runtime checks and logging. We describe the usage steps of Diagnosys for testing and debugging a given kernel-level service.

Debugging benefits. We present an assessment of our approach using both qualitative and quantitative measurements. Qualitatively, we show that the log of dangerous operations recorded by Diagnosys contains reliable information about the crash of the kernel. The experiments for this evaluation has consisted on replay real faults that were reported to kernel developers. We quantitatively evaluate the improvement of Diagnosys on the debugging of kernel-level services. We have used in this case mutation testing on 10 kernel-level services from various categories, including both device drivers and file systems.

Limited overheads. We measure the overhead introduced by Diagnosys runtime checks and logging operations. We also evaluate the impact of Diagnosys on service performance based on the stressed execution of two services. The experiences on a Gigabit Ethernet device driver as well as on a network file system reveals that the performance degradation is minimal.

1.3 Overview of the document

This thesis is organized in three parts: (I) the presentation of the background including the definition of the kernel-level service development problem and the state-of-the-art of relevant research work, (II) the description of the approach that we propose to support kernel developers, and (III) a discussion on the evaluation of the Diagnosys tool.

Background. The scientific background of our work is highlighted in Chapters 2 through 5. In Chapter 2 we provide an overview of monolithic OSes to highlight their issues with fault tolerance, with a case study on Linux. We also briefly discuss in this chapter the increasingly widespread core/plug-in architectures and further expose the difficulties in avoiding API usage bugs. Chapter 3 presents related work and discusses current techniques for addressing OS faults. In Chapter 4 we introduce the approach that we advocate for more readily supporting kernel developers. Finally, Chapter 5 details how the static analysis required by our approach is performed.

Proposed approach. The second part of this work details the proposed approach. We detail in Chapter 7 the characterization of safety holes and the static analysis performed to detect them in kernel code. Chapter 8 describes the design and implementation of Diagnosys for the Linux kernel. Finally in Chapter 9 we present the steps in using Diagnosys for testing and debugging a kernel-level service.

Assessment In the last part of this thesis, we assess the implementation of the Diagnosys approach. In Chapter 10, we perform experiments, which involve mutation testing, to quantify the debugging benefits provided by Diagnosys. We assess in Chapter 11 the performance overheads incurred by debugging interfaces on service execution.

Eventually, we conclude this thesis in Chapter 12. We summarize the contributions and offer a perspective for this thesis.

Part I

Background

Chapter 2

Monolithic Operating Systems

"If anyone had realized that within 10 years this tiny system that was picked up almost by accident was going to be controlling 50 million computers, considerably more thought might have gone into it."

Andrew S. Tanenbaum (*talking about MS-DOS*)

Contents

The term *operating system* is often used with two different meanings [Ker10]. Broadly, it is used to denote the entire package consisting of the core software managing a computer's resources and all of the accompanying standard software tools (e.g., file utilities, editors, graphical user interfaces, command-lines interpretors, etc.). More narrowly, the term refers to the kernel, i.e., the central software that manages and allocates the computer resources, including the CPU, the RAM memory, and devices. In this thesis we are concerned with this second meaning where *operating system* refers to the *kernel*. We explore in this chapter the principles of commodity operating systems and raise some challenges that OS service developers face.

2.1 Families of operating systems

Computers can run without an operating system. Appliances such as micro-ovens and washing machines contain computer chips where a single repetitive task is programmed to be executed once a button is punched. An OS however is meant to greatly simplify the writing and use of other programs, and to bring power and flexibility by providing a software layer to efficiently manage a computer's resources. There are grossly two architectures of operating systems, as depicted in Figure 2.1, that are essentially differentiated by how much of the OS services run in kernel mode (with the privileges of directly accessing all hardware and memory in the system).

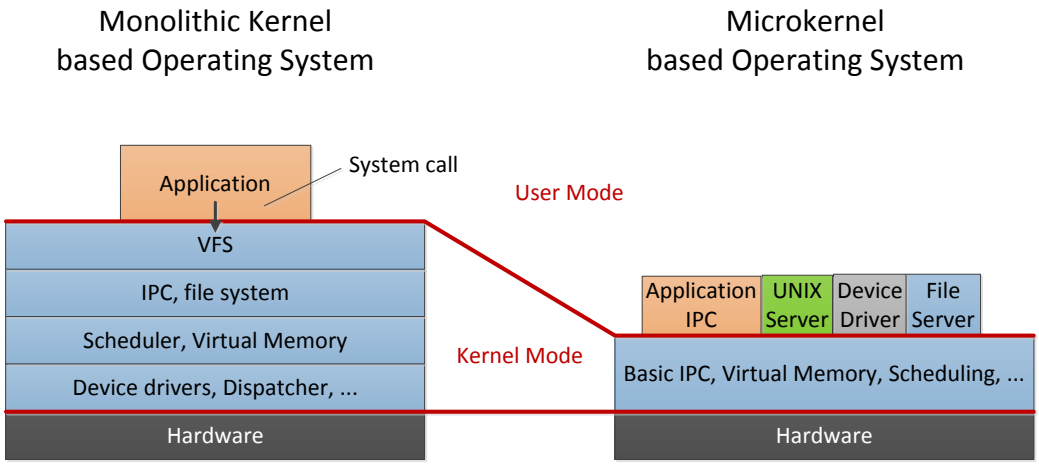

Figure 2.1: Main OS architecture models

Monolithic OSes. In Monolithic OSes, the entire kernel is reduced to a single binary containing the process management, memory management, file system, device drivers and the rest of services. Most older OSes are monolithic, including UNIX, MS-DOS, VMS, MVS, OS/360, MULTICS, and many more.

Microkernel-based OSes. Microkernels are an alternative to monolithic kernels. In a microkernel, most of the OS services run as separate processes. Most of these processes also run outside the kernel, the actual kernel being only constituted with the near-minimum software for handling message handling among kernel-level services, interrupt handling, low-level process management, and I/O. Examples of microkernel-based OSes include ChorusOS, Mach, Minix and Singularity.

The microkernel architecture was designed to address the fast-paced growth of kernels and the challenges that they brought along. Microkernels are supposed to ease the management of OS code, bring more security and stability by reducing the amount of code running with full privileges, i.e. in kernel mode. Monolithic architectures have long been preferred to first generation microkernels because of performance reasons. However, second generation of micro-kernels have resolved this issue and several research work have provided evidence showing that microkernel systems can be just as fast as monolithic systems [CB93, HHL+97, Ber92].

Nonetheless, today's commodity OSes embark a monolithic kernel architecture, except Windows NT which was influenced by the Mach microkernel. In the context of this thesis we focus

our contributions to helping developers of services for such OSes in their testing and debugging tasks. We first revisit the principles of monolithic OSes to highlight the difficulties facing both users and developers of kernel-level services. In the second part of this chapter, we briefly introduce our case study, the Linux kernel.

2.2 General principles of monolithic OSes

An OS is constituted of many independent parts, each providing simple individual features. A monolithic kernel as represented in Figure 2.2 concentrates all these parts in one (possibly large) system running with full privileges. Aside from low-level functionalities such as InterProcess Communication (IPC) and CPU scheduling, these parts also include the variety of device drivers, file system implementations, most instances of network protocols, etc.

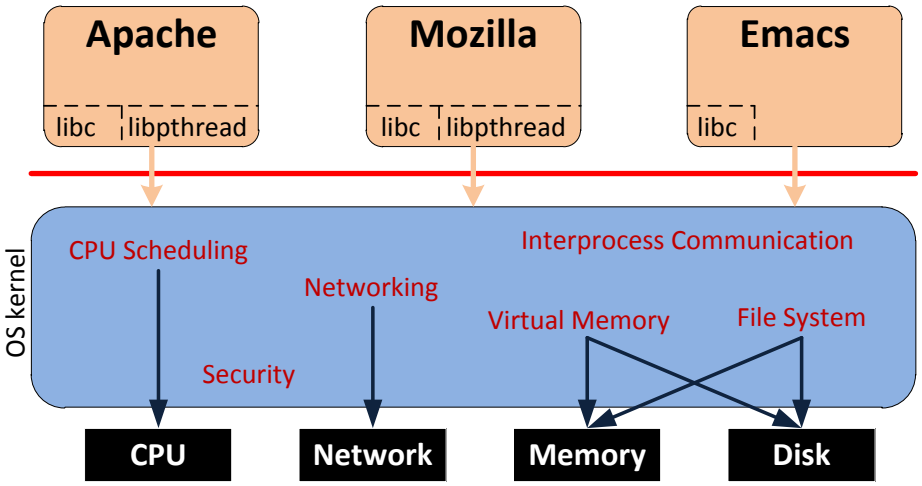

Figure 2.2: Architecture of Monolithic kernel-based systems

There are various privileges granted to the kernel for accessing otherwise protected resources such as physical devices or application memory. However, the main characteristic in the design of monolithic kernels is that no attempt is made to restrict the privileges to the different services in the kernel:

- The entire OS, i.e., all kernel-level services, executes in kernel mode, with maximum privileges. There is thus a great potential for both accidental and malicious misuse of privileges.
- There is no separation in the implementation of kernel parts. Any service can use the functionality of another. Furthermore, any service can access all memory including memory that is being used by others.
- New services accumulate in the kernel and participate in substantially increasing its size.

Numerous peripheral devices and file system architectures are regularly unveiled. Any monolithic kernel would therefore be expected to include all possible anticipated functionality, in form of drivers, already compiled into the kernel object. All services would then be loaded into memory while they may not be used during the uptime of the computer. This scenario, which is obviously not bearable,

has lead OS designers to resort to a plug-in architecture where services can be loaded and unloaded on-the-fly

2.3 Plug-in software architectures

Monolithic operating systems are exposed to well-known challenges in the practice of software engineering, particularly to challenges related to the constant increase of functionalities. To cope with such challenges, modern successful software projects have adopted a plug-in architecture model where the code is centered around a core for which clients should write plug-ins.

In the plug-in model, a core code is structured such that certain well-defined areas of functionality can be provided by an external module – the plug-in. Plug-ins can therefore be written and compiled separately from the core, and typically by other developers who are not involved in the development of core code. Plug-ins, while adding functionalities, often require, for their own functioning, some functionalities of the core or of other plug-ins. Examples of software implementing a plug-in model are widespread in the software engineering community: the Eclipse integrated development environment, the Firefox web browser, the Thunderbird email client, or the Microsoft Office word processor.

The plug-in model addresses a number of issues in software engineering and provides various advantages. We briefly summarize the most salient reasons why applications support plug-ins:

- First, plug-ins allow to more readily add features to an already large code base. This property of software plug-ins is praised by both users, who wish to choose the capabilities of the software, and developers who wish to extend them, in a limitless way.
- Second, project architects often propose to break down a software architecture into multiple components that can be plugged in a minimal core. This procedure allows to reduce the size of the original application.
- Third, the plug-in model allows for the interaction of various code parts developed under different licensing schemes. Plug-ins indeed separate source code and as such do not threaten the liability of the core application.
- Finally, adoption of the plug-in model enables third-party developers to participate in the growth of a software project by providing new abilities that extend the core code base.

The reasons outlined above are valid to any successful monolithic operating system. Indeed, the need for new drivers to support new manufacture devices suggest a need for controlling the growth of the kernel. Furthermore, device manufacturers who are not necessarily taking part to the development of the kernel may need to separately develop their code and plug-it to the kernel. Finally, introducing modularity into the implementation of the OS may ease maintenance. Loadable kernel modules were therefore imagined to address these issues by re-using a plug-in model for kernels.

2.3.1 Loadable Kernel Modules

While the architecture of most contemporary kernels is monolithic, their internal implementations provide a mechanism for dynamically loading code for extending an already-running kernel. This modular implementation is common in modern monolithic OSes, including most current Unix-like systems and Microsoft Windows. The extension modules, which are referred to as *Loadable Kernel Modules* (LKMs) or more simply as Kernel MODules (*KMOD*), are typically used to add support for new hardware and/or file systems.

In FreeBSD, LKMs are denoted *kld*. In OS X, they are referred to as kernel extensions (*kext*). In any case, LKMs, when they are loaded, are part of the kernel. They communicate with the *base kernel* in the same way as the other parts of the kernel that were already bound, at compile time, into the image that is booted. When the functionality provided by an LKM is no longer required, it can be unloaded to reduce the runtime memory footprint of the OS.

As an extension code that must interlock with an existing code base, a plug-in is required to properly use the interfaces available for its interactions with the core. In the case of a monolithic kernel where the loadable kernel module, once loaded, is entirely part of the kernel and runs with privilege mode, any misuse of the kernel API will lead to an unrecoverable failure that can bring about the demise of the entire system.

2.3.2 API usage challenges

An Application Programming Interface (API) specifies how different components of a software communicate and interact. In the plug-in model, the API is the central interface that lists the set of routines/functions/methods that are accessible from the core code base to the plug-ins. A specific class of problems can then be found in the usage of API functions due to implicit usage protocols. In these cases, developers are in presence of core code that in itself is not actually wrong, but that is prone to errors when combined with certain kinds of plug-in code. Indeed, APIs are subject to misuses as illustrated in the example of Figure 2.3.

```
unsigned char *skb_put(struct sk_buff *skb, ...)
{   unsigned char *tmp = skb_tail_pointer(skb);
    SKB_LINEAR_ASSERT(skb);
    skb->tail += len; ...
}
```

a) Unrobust core API function

```
tx_skb = dev_alloc_skb(pkt_len);
pkt_data = skb_put(tx_skb, pkt_len);
```

b) Buggy plugin code

Figure 2.3: Bug fix of the usage of Linux `skb_put` API function

We consider an example of function, *skb_put*, that is part of the Linux core and that can be used for adding data to a network `skb` buffer. An excerpt of the code is shown in Figure 2.3(a). The *skb_put* function begins with some operations that extend the tail pointer of the network buffer. In particular the first line of code calls a helper function that retrieves the current value of this tail pointer. *skb_put* is not robust in that it assumes that its argument is not NULL.

Figure 2.3(b) shows some plug-in code, i.e., a network driver, that calls the *skb_put* function in an unsafe way. In particular, it calls the function with the first argument `tx_skb` that it has obtained from a call to the function *dev_alloc_skb*. However, *dev_alloc_skb* can return NULL in some cases, which will then lead to a dereferencement fault in the function *skb_put*.

The illustrated example details a situation where the plug-in code is actually buggy, because it is its responsibility to check for NULL. This is combined with a case where the core code is not robust, because it dereferences its argument without checking for NULL. Consequently, the bug in the plug-in code leads to a failure within the core code. Moreover, in this example, the failure is not within the code called by the plug-in, but, deeper, in a function that this code calls, making the problem harder to spot. We acknowledge however that the core is not actually wrong, because it can put whatever preconditions it wants on its callers. For example, in Linux, the standard practice is to avoid inserting validity checks for the arguments of API functions for performance reasons. Indeed, inserting checks may lead to costly double checks, both by the calling code and in the function definition, which could

introduce an important performance penalty for functions that are called a significant number of times during a service execution. In our experiments with the *NFS* file system, when transferring eight (8) GB of data in less than 10 minutes, we have recorded over sixteen (16) millions calls to API functions, in which validity checks were justifiably omitted. Nonetheless, we refer to the problem posed by such an implementation as a ***safety hole***, because the core function can be used in a context that will make the combination of the plug-in and the core a bug.

The *safety hole* issue is exacerbated in the plug-in model by two main modern software engineering facts:

- Core code, which can amount to millions of lines of code, is often not well known to the plug-in developer, making it difficult to debug safety hole-related problems.
- API functions are not properly documented, making it more likely for plug-in developers to misuse them, especially when the usage preconditions are idiosyncratic.

In particular, in monolithic OSes that use a mechanism of loadable kernel modules, and that are developed as open source projects, the issues with safety holes is significantly exacerbated. Indeed, such OS core code size makes them some of the biggest project code base, while the number and disparity of developers makes for the most distributively developed projects.

2.4 Linux case study

The monolithic kernel family includes most of contemporary operating systems including Linux and Windows[1]. In this thesis we focus our study, including the implementations and experiments, on the Linux open-source kernel.

Linux is a monolithic multi-user multi-task OS built in the model of Unix. First released in October 1991 by Linus Torvalds, Linux has grown to be maintained by over 9,000 kernel developers around the world and many more bug reporters and patch submitters who have contributed to the 13 millions lines of code that are now available in the version 3.6 released in September 2012.

Linux was originally developed for Intel x86-based PCs, but has since been ported to more computer hardware platforms than any other operating system [IBM01]. It is a leading operating system in servers, mainframe computers and supercomputers, and is also widespread in embedded systems. Countless distributions of Linux are available with different focus on security, performance, user experience, etc. As of 2012, we have been able to list about 500 known distributions.

2.4.1 Development model

The Linux kernel development model process is based on the assumption that the source code of all kernel-level services is available within the publicly available kernel source tree, and thus kernel APIs are, for efficiency, as robust as required by their internal client services. Furthermore, kernel developers can freely adjust the kernel APIs, as long as they are willing to update all of the affected service code. The kernel is thus, by design, maximally efficient and evolvable, enabling it to rapidly meet new performance requirements, address security issues, and accommodate new functionalities. Unfortunately, these assumptions complicate the task of the developers of new services who require more safety and help in debugging.

[1] Except Windows NT

To cope with the sustained support of devices and the interest of developers, Linux has adopted a modular architecture allowing kernel-level services, such as device drivers and file systems, to be developed outside the mainline tree. Developers however face the challenges of writing code against the Linux core since it does not provide a well-defined API nor a stable one [KH].

2.4.2 Kernel exported functions

The Linux kernel provides support for conveniently loading an out-of-tree driver into a running kernel using the loadable kernel module facility [dGdS99]. Such a module may interact with the kernel via those kernel functions that the Linux kernel developers have chosen to export using the `EXPORT_SYMBOL` mechanism. To produce a working driver, a developer must use these functions according to the correct usage protocol. Nevertheless, there are many exported functions and their usage protocol is often idiosyncratic, making correct usage of these functions a challenge for developers. We now examine how kernel interfaces have evolved in recent releases and how this evolution impacts their usage.

2.4.2.1 API evolution

Figure 2.4 presents the number of functions exported by versions of the Linux kernel released over the course of three years (December 2008-January 2011). The graph distinguishes between the functions already exported in the first considered version, Linux 2.6.28, and the functions added or modified since then. The results show that the number of exported functions has been steadily increasing. For example, Linux 2.6.37 exports 224 more functions than Linux 2.6.36. The results also show that the set of exported functions continuously changes [KH]. Indeed, over 3,000 (25% of all) exported functions present in 2.6.37 were not available in 2.6.28, while over 1,000 (10% of all) exported functions present in 2.6.28 have disappeared. In addition, over 3,000 (33% of all) functions present in 2.6.28 have been modified in 2.6.32.

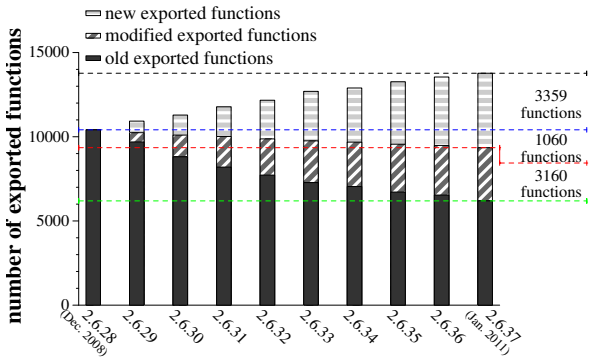

Figure 2.4: Evolution in the number of functions exported by the Linux kernel (baseline Linux 2.6.28)

2.4.2.2 API documentation

One way to become aware of the usage protocols of the functions exported by the Linux kernel is to read the associated documentation. A small number of functions exported by the Linux kernel are documented in Section 9 of the Linux "man" pages, if these are installed by using the appropriate kernel make file target. Figure 2.5 shows that the number of interface functions with man pages is

increasing, but at a slightly lower rate than the number of interface functions themselves. Furthermore, less than 15% of the exported functions have man pages available in the versions considered. Finally, the second curve (dashed line) reveals that the documentation of exported functions does not significantly improve over time.

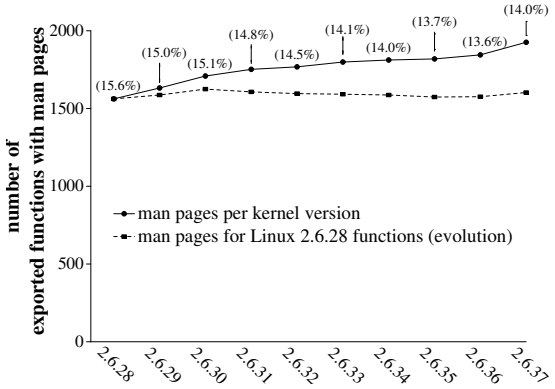

Figure 2.5: Number of kernel interface functions with man pages (percentages are relative to the total number of exported functions in the kernel version)

2.4.2.3 API usage examples

In the absence of documentation, a common programming strategy is to find reliable examples of how exported functions are used. Indeed, it is well known that driver developers often copy/paste code fragments of kernel code [LLMZ04]. Yet, as illustrated in Figure 2.6 for recent releases, an increasing number of exported functions are no longer used by in-tree code, thus depriving programmers of usage examples. Actually, some of these functions are not used by in-tree code because their use is no longer recommended. For instance, the *sys_open* and *sys_read* functions that implement the *open* and *read* system calls have been exported for a long time, even though opening and reading files at the kernel level is rarely appropriate [KH05]. Attempts to remove such functions in order to prevent misuse by third-party developers have lead to heated discussions as these functions may still be required by out-of-tree code.[2]

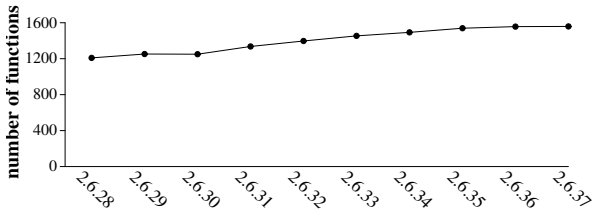

Figure 2.6: Number of kernel exported functions not used by in-tree C code

We look in detail at the exporting and use of exported functions in the *drivers* directory, as this directory contains code that is most similar to the code found in out-of-tree kernel-level services. Figure 2.7 shows that almost 70% of the exported functions used by drivers are provided by other drivers and the many libraries implemented in the drivers directory. The invocations of these interface

[2] http://lwn.net/Articles/249265

functions constitute about 20% of all the usage calls of interfaces in *drivers*, suggesting that drivers use more functionalities from other types of services, but still require a significant amount of interactions with other drivers. Overall, the figures suggest that drivers in general interact to a great extent with code in this directory. Unfortunately, as device drivers evolve and are adapted fast, code in the drivers directory is often revisited, potentially introducing changes in the interfaces proposed by the code. As a result, out-of-tree code which interact with code from this directory is likely to fail to follow the frequent updates in the conventions for accessing kernel interfaces.

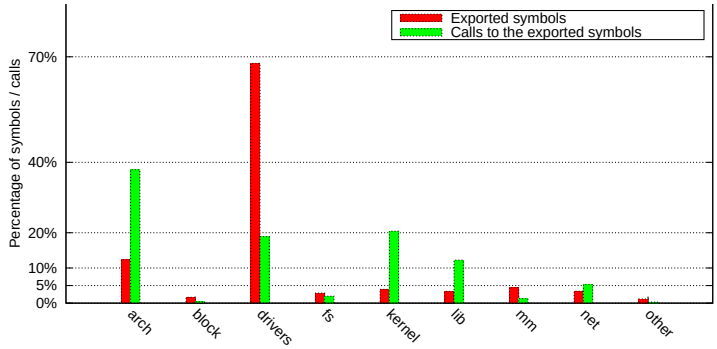

Figure 2.7: Number of kernel exported functions not used by in-tree C code

2.5 Summary

In this chapter, we have introduced monolithic operating systems with a focus on the Linux kernel. We discuss some of the challenges for developing kernel-level services in this context, in particular with regards to the use of the kernel APIs.

In monolithic OSes, a single fault in a kernel-level service may bring about the demise of the entire system without leaving any clue as to what went wrong. Some of those faults are due to misuses of kernel API functions which are largely undocumented and contain implicit usage preconditions.

We have described a study of the Linux kernel programming interface to explore how API functions evolve and why they constitute a source of difficulties for kernel-level service developers. We have thus shown that the interface is large and ever changing, and that API functions are scarcely documented, often with limited usage examples for novice programmers. Theses characteristics of the kernel interface combined with the idiosyncratic implementations of API functions create a significant number of opportunities for violating different usage preconditions.

Chapter 3

Dealing with System Failures – Testing, Debugging, Recovery –

"The Linux philosophy is to laugh in the face of danger.
Oops. Wrong one. Do it yourself. That's it."
Linus Torvalds

Contents

Operating systems fail. These failures are often due to programming errors that manifest themselves in different ways. A large body of research work has been proposed to characterize OS faults and to address them. In this chapter, we discuss the origin and the manifestations of OS failures and investigate related work for dealing with kernel reliability.

3.1 Kernel failures

When the execution of kernel code stumbles upon a faulty code path, OS failure may ensure. The manifestations of these failures can be classified in two categories, i.e., kernel crashes and kernel hangs, but the immediate solution to any of them generally involves rebooting the entire system to reinitialize the failed service.

3.1.1 Kernel crashes

When the execution of the kernel encounters a fault, a kernel oops is issued to warn about the failure. When an oops occurs, the kernel may be able to continue operating, although the kernel service involved in the oops will no longer be available. However, often, kernel oops lead to kernel panics after which the system cannot continue running and must be restarted. In device drivers, kernel oops typically don't immediately cause panics, however they often leave the system in a semi-usable state. Nevertheless, from the developer point of view, a kernel oops also requires a system reboot for re-testing the failed module. In some cases, a kernel oops may cause a kernel panic when something vital is affected. In this thesis, the expression "kernel crash" is used to refer to the state of the kernel when its execution leads to a kernel oops or a kernel panic.

Kernel oopses are caused, for example, by the kernel dereferencing an invalid pointer. The equivalent of such fault for user-space programs is known as the segmentation fault from which the program cannot recover. Figure 3.1 shows an excerpt of a kernel oops message which contains important information for debugging as produced by the Linux *kerneloops* program. First, the oops displays the type of error that occurred, in this case "`Unable to handle kernel NULL pointer dereference at [...]`", indicating the reason for the crash. Then, follows information about the oops number which indicates the number of the oops so that the developer may know the order at which multiple oopses have occurred in order to select the first one, as the most reliable, for back-tracing the error. The code segment and the address of the instruction that were being executed are indicated by the EIP field value. Finally, the *kerneloops* program prints the contents of the CPU's registers and a stack backtrace, i.e., the list of functions that the crashed process was in when the oops occurred. Developers are required to rely on the Symbol Map to map the function names to the numeric addresses present in the call trace.

It is to be noted that a kernel crash systematically leads to the service whose process has failed to be unresponsive. It does not however necessarily lead to the demise of the entire system. For example, if the crash occurs in a sound driver and in code exclusively related to that driver, the sound system will probably stop functioning but the computer will likely continue running (with compromised reliability). Nevertheless, when the crashed code is vital to the system, the first oops may be followed by other oops from other processes and this cascade of oops can lead to a kernel panic, forcing a reboot of the entire OS.

3.1.2 Kernel hangs

Kernel hangs are OS failures that leave the system unresponsive, and systematically require a system reboot. They manifest themselves when a process get stuck in its execution without giving the possibility for other processes to benefit from execution time. Kernel hangs are generally linked to issues with locking and interrupt management.

Deadlocks are a common programming error. In this case, a developer has misplaced lock acquisition and release statements for protecting critical sections, causing the possibility for processes to indefinitely wait for a lock to be released, or the possibility for processes to attempt to release locks that they have not acquired themselves.

In the Linux kernel, interrupts are used as a means for hardware, such as sound, video and usb devices, to interrupt whatever the CPU is currently doing and execute some special code for handling a new event. This special code usually operates in the CPU *interrupt mode* during which no other interrupts can happen, with some exceptions, for example when the CPU ranks the interrupts. Handling interrupts however is a sensitive task as there are regions in the kernel which must not be interrupted

```
1  [43470.712897] BUG: unable to handle kernel NULL pointer dereference at 0000010c
2  [43470.712900] [...]
3  [43470.712909] Oops: 0000 [#1]
4  [43470.712911] [...]
5  [43470.712913] Modules linked in: nls_utf8 isofs usbhid hid [...]
6  [43470.712943] Pid: 31941, comm: mount Tainted: [...]
7  [43470.712946] EIP: 0060:[<c01fc71f>] EFLAGS: 00010282 CPU: 0
8  [43470.712948] EIP is at iput+0xf/0x60
9  [43470.712950] EAX: ffffffff4 EBX: ffffffff4 ECX: fffebedc EDX: 00000000
10 [43470.712951] ESI: f717de00 EDI: 00000000 EBP: f287bde0 ESP: f287bddc
11 [43470.712953]  DS: 007b ES: 007b FS: 0000 GS: 00e0 SS: 0068
12 [43470.712955] Process mount (pid: 31941, ti=f287a000 task=f1efe500 task.ti=f287a000)
13 [43470.712956] Stack:
14 [43470.712958]  0000002f f287be78 f84f50e7 f84fab68 00000003 ffffffff ffffffff 00001a00
15 [43470.712961] <0> 00000000 f1829000 f390c200 756ebd83 00000800 01400100 000003e8 000003e8
16 [43470.712965] <0> f37d31b0 ffffffff ffffffff f37d31b0 dec68000 dec68000 00000000 fffffffea
17 [43470.712969] Call Trace:
18 [43470.712973]  [<f84f50e7>] ? isofs_fill_super+0x70e/0x8c4 [isofs]
19 [43470.712977]  [<c031b09a>] ? snprintf+0x1a/0x20
20 [43470.712981]  [<c01ecc12>] ? get_sb_bdev+0x162/0x1a0
21 [43470.712984]  [<f84f49d9>] ? isofs_fill_super+0x0/0x8c4 [isofs]
22 [43470.712988]  [<f84f631c>] ? isofs_get_sb+0x38/0x40 [isofs]
23 [43470.712991]  [<f84f49d9>] ? isofs_fill_super+0x0/0x8c4 [isofs]
24 [43470.712994]  [<c01ec777>] ? vfs_kern_mount+0x67/0x170
25 [43470.712997]  [<c01ff1c3>] ? get_fs_type+0x33/0xb0
26 [43470.712999]  [<c01ec8de>] ? do_kern_mount+0x3e/0xe0
27 [43470.713002]  [<c0201c00>] ? do_mount+0x430/0x640
28 [43470.713005]  [<c01b746b>] ? __get_free_pages+0x2b/0x30
29 [43470.713008]  [<c01ffcb1>] ? copy_mount_options+0x41/0x140
30 [43470.713010]  [<c0201e7b>] ? sys_mount+0x6b/0xa0
31 [43470.713014]  [<c0108583>] ? sysenter_do_call+0x12/0x28
32 [43470.713015] Code: ff ff 8b 4b 4c 39 4b 5c 8d 74 26 00 0f 89 41 ff ff ff 66 [...]
33 [43470.713033] EIP: [<c01fc71f>] iput+0xf/0x60 SS:ESP 0068:f287bddc
34 [43470.713036] CR2: 000000000000010c
35 [43470.713038] ---[ end trace 711cf1989f2fc2e0 ]---
```

Figure 3.1: Excerpt of an example of Kernel oops report in the *isofs* filesystem

at all. For these cases, kernel developers can use a pair of available interrupt handling primitives to delimitate such regions. Unfortunately, mistakes in the use of such primitives can be fatal to the system because they can leave the system uninterruptible, i.e., no hardware can function correctly, and lead to a kernel hang.

3.2 Faults in Kernel

Currently, most operating systems are written in the C native programming language for performance reasons. The C language is however known to be less safe than new generation languages, such as objected oriented programming languages, and its issues also contribute to degrade kernel reliability. Furthermore, OS code is complex and involves various idiosyncrasies that only experienced core developers can thrive with. For example, in the Linux kernel, function pointers are pervasive, and are the source of different programming errors whose diagnosis is challenging. A 2001 empirical study on operating system errors by Chou *et al.*, based on systematic analysis of kernel source code with the Coverity tool, has provided many insights on programming errors that kernel developers leave in the Linux kernel[1] code [CYC+01]. The authors of this study have found that device drivers have

[1]The study was based on kernel release 2.4.1

Category	**Actions to avoid faults**
Block	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held
Null	Check potentially NULL/ERR_PTR pointers returned from routines
Var	Do not allocate large stack variables $(> 1K)$ on the fixed-size kernel stack
INull	Do not make inconsistent assumptions about whether a pointer is NULL/ERR_PTR
Range	Always check bounds of array indices and loop bounds derived from user data
Lock	Release acquired locks; do not double-acquire locks
Intr	Restore disabled interrupts
Free	Do not use freed memory
Float	Do not use floating point in the kernel
Real	Do not leak memory by updating pointers with potentially NULL realloc return values
Param	Do not dereference user pointers
Size	Allocate enough memory to hold the type for which you are allocating

Table 3.1: Categorization of common faults in Linux [CYC+01, PLM10]

error rates up to three to seven times higher than the rest of the kernel. They also found that the largest quartile of functions have error rates two to six times higher than the smallest quartile and that the newest quartile of files have error rates up to twice that of the oldest quartile. This last finding confirms that kernel code "hardens" over time. Finally, they have found, at that time, that bugs remain in the Linux kernel an average of 1.8 years before being fixed.

Ten years later, Palix *et al.* [PST+11] have reproduced the study on recent versions of the Linux kernel using the Coccinelle open-source static analyzer. They found that, though code in the drivers directory have improved in quality, they still contain the highest error rates. Furthermore, according to their study on multiple versions of Linux, each release of the Linux kernel contain between 600 and 700 faults. These faults mainly involve memory allocation issues, pointer dereferencements, locking and interrupt management. We summarize the different categories of common faults in Linux in Table 3.1.

3.3 System Failures: What To Do About Them?

In previous sections, we have shown that kernel code contains various categories of programming errors that compromise the reliability of operating systems. Various research directions have been followed to address the issue. These include building tools for testing software systems, improving the resilience of kernel code to OS errors, automating the production of bug-free software, identifying bugs in developer manually-written code, improving debugging by providing better failure information, and improving programming habits.

3.3.1 System robustness testing

To address the presence of such errors, developers often submit their code to a battery of tests designed to stress the OS and assess its robustness. To this end, fault injection has been applied to the

Linux kernel to evaluate the impact of various fault classes: Albinet *et al.* have proposed a framework for characterizing the impact of faulty drivers on the robustness of the Linux kernel [AAF04]. Cotreno *et al.* have focused on injecting faults for detecting hangs in Linux [CNR09]. Marinescu and Candea [MC11] focus on the returns of error codes from userspace library functions. Overall, these research contributions have provided evidence that puts into question the robustness of the Linux kernel. However, they do not provide explanations on the root causes of their observations in order to allow developers to address them early. Furthermore, the test suites proposed by the authors are generally restricted to a few, though the most common, categories of faults, leaving many others of the common faults that we have summarized in Figure 3.1.

> Frameworks for assessing OS robustness provide insights on the quality and reliability of the kernel. However, they often target a niche of faults and are not necessary useful for producing explanations of the observations made based on their outputs. Yet, in the early stages of driver development, kernel developers need more information on the origins of their errors.

3.3.2 Automating OS software production

Having established, through extensive robustness testing, that current monolithic systems present reliability issues, researchers have developed a variety of approaches to improve the process of developing driver code. The aim of these research work has been to automate as much as possible the production of driver code in order to avoid, as much as possible, programming errors. For example, Merillon *et al.* have proposed the Devil language-based approach for allowing developers to specify a device driver that will be automatically generated [MRC+00]. With Dingo [RCKH09] and Termite [RCK+09], Ryzhyk *et al.* also propose to use specifications for automatic driver synthesis. Finally, Chipounov and Candea propose RevNic [CC10] for deriving new drivers from existing drivers for other operating systems.

> Research work from academia for automating the generation of driver code has proven to be very effective in producing efficient and reliable code. However, one main weakness of the proposed approaches is that they are not broad enough as the built tools only work for narrow classes of device drivers. Thus, the approaches still face resistance to be adopted in industry where the standard practices remain to write device drivers by hand.

3.3.3 Recovering from OS failures

OS code is bound to contain programming errors, especially in device drivers. To prevent driver defects from compromising the entire kernel against which the drivers run, a number of approaches have been developed for isolating drivers [CCM+09, **?**] or for recovering from OS failures [SBL03, MSZ09]. Swift *et al.* have proposed Nooks [SBL03], a framework whose aim is to improve the reliability of operating systems, by implementing a layer that enables the OS to recover from device driver crashes. Zhou *et al.* have also proposed SafeDrive [ZCA+06] that relies on language-based techniques to add safety in the execution of device drivers. TwinDrivers [MSZ09], proposed by Menon *et al.*, also aim at providing a safe environment for executing device drivers. Finally, the DD/OS [LUSG04] system proposed by Levasseur *et al.* relies on virtual machines to isolate device drivers from the rest of the kernel.

> Approaches for protecting the kernel from driver defects have been successful in research venues a decade ago. Their adoption however remains only for research purpose. In production settings, as the authors have later admitted, the proposed solutions are too heavyweight.

3.3.4 Logging

An immediate debugging technique for diagnosing OS failures is to navigate through runtime logs to identify lines reporting events that may explain OS misbehaviors. Unfortunately, runtime logs are frequently insufficient for failure diagnosis especially in case of unexpected crashes [YMX+10, YPH+12]. A research avenue has then been to improve runtime logging mechanisms so as to benefit from more exploitable information when failures occur. Yuan *et al.* have recently proposed *LogEnhancer* [YZP+] to enrich log messages with extra information. *LogEnhancer* does not however create new messages in places where they might be necessary.

> Runtime logs remain the most readily available source of information on system failures. Kernel code however only includes logging instructions in places that core developers believe to be important for monitoring functionality. Other places, including interaction sites as at the boundary between the kernel and a driver, are not monitored enough, missing important information that driver developers could find useful during driver testing.

3.3.5 Static bug finding

Model checking, theorem proving, and program analysis have been used to analyze OS code to find thousands of bugs. Ball *et al.* have designed a static analysis engine that was incorporated into Microsoft's Static Driver Verifier (SDV) tool [BBC+06] where it is used for finding kernel API usage errors in a driver. Post and Küchlin have manually extracted conformance rules from the Linux documentation and have formulated them in the SLIC extended version of SDV's SLIC specification language [PK07]. Dawson *et al.* [ECH+01] have proposed techniques to automatically infer checking information from the source code itself, looking for contradictions between the code and what appears to be the programmer beliefs. For example, a call to "spin_lock" followed once by a call to "spin_unlock" implies that the programmer may have paired these calls by coincidence. If the pairing happens 999 out of 1000 times, though, then it is probably a valid belief and the sole deviation a probable error. Unfortunately, these tools take time to run and the results require time and expertise to interpret. Thus, these tools are not well suited to the frequent modifications and tests that are typical of initial code development.

Numerous approaches have proposed to statically infer so-called *protocols*, describing expected sequences of function calls. Thus, in the search for contradictions, Dawson *et al.* were actually inferring call protocols first [ECH+01]. With their "What You See Is Where It Bugs" approach [LBP+09], Lawall *et al.* rely on the Coccinelle static analysis tool to declaratively specify the protocols and identify their violations in driver code. Li and Zhou have proposed PR-Miner [LZ05] for identifying, automatically, implicit programming rules in large code base such as the Linux kernel. Ramanathan *et al.* have proposed Chronicler [RGJ07], a tool that applies scalable inter-procedural path-sensitive static analysis to automatically infer accurate function precedence protocols. In general, these approaches have focused on sequences of function calls that are expected to appear within a single function, rather than the specific interaction between a service and the rest of the kernel.

The use of advanced type systems have been demonstrated to be efficient in eliminating some unsafe usage preconditions in API functions. For example, Bugrara and Aiken propose an analysis to differentiate between safe and unsafe userspace pointers in kernel code [BA08]. They focus, however, on the entire kernel, and thus may lead to the analysis of parts of code that do not interest a given service developer who is only familiar with his own code.

> In general, static bug finding tools are not intensively used by driver developers before the code is shipped to users. Indeed, running these tools take some time and produce a significant number of false positives which usually deter developers. This situation is mainly due to the fact that the analysis performed in these cases are not focused to specific code places, such as kernel interfaces, but rather runs through function calls and definitions across the entire OS.

3.3.6 Implementing robust interfaces

To prevent API usage errors from seriously impacting the reliability of software systems, different approaches have been developed for strengthening the robustness of interfaces. LXFI [YHD+] isolates kernel modules and includes the concept of *API integrity*, which allows developers to define the usage contract of kernel interfaces by annotating the source code. LXFI, however, aims at limiting the security threat posed by the privileges granted to kernel modules, while various other categories of common faults in kernel code still have the potential of compromising the integrity and the reliability of operating systems.

Healers automatically generates a robust interface to a user-level library without access to the source code [FX03]. It relies on fault injection to identify the set of assumptions that a library function makes about its arguments. Healers can obtain information about runtime values, such as array bounds, that may be difficult to detect using static analysis. However, Healers does not address the important safety holes involving locking behaviors since they require calling-context information. Indeed, supporting locking safety holes would require testing the state of all available locks, which would be expensive and are likely unknown.

> Approaches for adding robustness layers to APIs aim at protecting the system from usage errors. However, they focus on specific classes of problems and are not designed to account for the variety of programming errors that non-malicious developers leave in their code. Furthermore, kernel programmers are wary of such layers which are known to generally degrade execution performance.

3.3.7 Programming with contracts

Since developers regularly misuse system APIs, especially when those include implicit and undocumented usage preconditions, researchers and practitioners have proposed to explicitly detail for each API function the *contract* of its usage to avoid misbehavior. A software *contract* represents the agreement between the developer of a component and its user on the component's functional behavior [FLL+02, HPSA10, Mey88, Mil04]. Contracts include pre- and post-conditions, as well as invariants. A safety hole is essentially the dual of a contract, in that a contract describes properties that the context should have, while a safety hole describes properties that it should not have.

Contract inference is analogous to the process inferring safety holes. Arnout and Meyer infer contracts based on exceptions found in .NET code [AM03]. Daikon infers invariants dynamically by running the program with multiple inputs and generalizing the observations [EPG+07]. Linux

kernel execution however is highly dependent on the particular architecture and devices involved, and thus a service developer would have to actively use Daikon in his own environment. Finally, from the Daikon invariant list,[2] we note that only a few invariants may be used to uncover relevant safety holes in kernel code. It is, for example, the NonZero invariant which may correspond to a safety hole relevant to faults that involve dereferencing invalid pointers. Other kinds of safety holes, including those relevant to memory release errors, user/pointer bugs and other kernel specific faults are not handled by Daikon.

The Extended Static Checker for Java (ESC/Java) [FLL+02] relies on programmer annotations to check method contracts. Annotation assistants such as Houdini [FL] automate the inference of annotations. Houdini supports various exceptions involving arguments, such as NullPointerException and IndexOutOfBoundsException, but does not provide tests for e.g., the validity of allocated memory.

> Implementation of software contracts aims at avoiding confusion in the use of APIs. Nonetheless, they require too much effort from developers who fail to even include proper documentation. Thus, contract description is not yet part of kernel developers habits. Contract inference on the other hand is limited to few rules and remain cumbersome for daily usage.

3.4 Summary

In this chapter, we have studied how programming errors compromise the reliability of operating system kernels such as Linux. We have then investigated research work that attempt to provide solutions for improving the safety of system code. Over the years, various research efforts have been directed at evaluating the impact of some classes of faults on the robustness of operating systems, at automating the production of more reliable, i.e., bug-free, OS code, at providing means to dynamically recover from OS failures, at enhancing logging mechanisms for improving failure diagnosis, at detecting bugs early in source code, and at strengthening the usage of API functions.

We have discussed the weaknesses of each approach and shown how, alone, the different approaches cannot fully resolve the problems of kernel reliability. The lessons learned from the success and failures of these approaches can however be leveraged to design a new approach for supporting kernel developers in the testing and debugging of kernel-level services in order to produce more reliable services that will interact correctly with the kernel.

[2] http://groups.csail.mit.edu/pag/daikon, Documentation, Sec. 5.5

Chapter 4

Supporting Linux Kernel-level Service Developers

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?"

Brian Kernighan

Contents

In previous chapters, we have unveiled different challenges that developers face in the development of kernel-level services. We have also discussed related work that support our main concern that driver development processes would still benefit from the design of new approaches and advanced tools to improve developers activities.

In this thesis, we focus on the debugging process of kernel-level services such as device drivers and file systems that are built outside the mainline kernel tree. Indeed, before any driver code can be integrated into the kernel tree where it can benefit from the expertise of more experienced developers, the code must first meet a number of requirements, both in terms of functionality and reliability. Thus developers of out-of-tree code, including many developers in hardware companies, such as Intel, who need to build drivers for their products to make them Linux compliant, are on their own in the early stages of driver development. One particular source of errors is in the usage of kernel APIs that contain implicit and ill-documented usage preconditions. In this chapter we outline the different points that need to be addressed in the road of building a framework for supporting kernel developers. We have listed four main points starting with a better understanding of kernel APIs (Section 4.1) and the integration of enhanced and more informative debugging information in APIs that may be the source

of faults (Section 4.2). In a second step, we describe the need for more adding checking instructions without compromising OS performance (Section 4.3). Finally, we discuss the recovery steps of crash information and the reliability issues that they are associated with (Section 4.4).

4.1 Thorough analysis of API safety holes

In a Plug-In software architecture model, APIs are a source-code-level means of communication between software components, namely core and plug-ins. API functions expose core functionalities to be used by external code. In kernel programming, APIs are mainly constituted of kernel functions that are available for use in kernel-level services. Unfortunately, API functions often contain idiosyncratic usage preconditions that introduce the potential for errors. Kernel developers are therefore challenged to acquire a substantial knowledge on the details of each API function before using them. In the Linux OS kernel, the API is further challenging to grasp due to a number of specificities in this open-source distributed development context.

4.1.1 Implementation idiosyncrasies

The Linux kernel internals include implementation details that are often missed by out-of-tree developers. For example, for performance reasons, kernel APIs are unrobust, by design, with limited checks on the validity of parameter pointer values. Critical sections may also be split with only one part implemented in the body of an API function, requiring calling functions to implement the other part. Thus, for example, an implicit usage precondition of an API function could be that it must be called with a specific lock held. There exists a variety of such usage preconditions and these lead to the proliferation of API safety holes as defined in Section 2.3.2 (p. 11).

The idiosyncratic details of kernel code is a challenge to kernel-level service developers who are not necessary experienced kernel software developers. In particular, code that is maintained outside the kernel tree does not benefit from the insights of experienced kernel developers.

4.1.2 Unstable API

The Linux kernel API is unstable and the kernel documentation contains an entire section where prominent kernel maintainers have discussed why the need for a stable Linux kernel seems like a non-sense for Linux [KH]. Functions are added and removed, at will, between version releases. Furthermore, kernel maintainers do not offer any guarantee on the stability of the design and implementation details of each function. For example, even the signature of API functions continuously change with parameters being added or removed. Finally, the results of our study previously illustrated in Figure 2.4 shows that the set of API functions continuously changes. Indeed, for example, over 3,000 (33% of all) API functions present in 2.6.28 have been modified in 2.6.32.

The unstable property of the kernel API makes it difficult for driver developers to keep up. Indeed, they are regularly required to check the implementation details of the functions that are called by their code, and potentially of all other core routines that are called in those functions. This exercise is particularly tedious for out-of-tree code that do not benefit from maintainers attention.

4.1.3 Undocumented API

In Section 2.4.2.2 (p. 13), we have shown that, in each release, only a small number of the Linux kernel API functions are documented. We have furthermore shown that documentation does not significantly

improve with time. Finally, existing documentation often misses to mention the usage preconditions that are otherwise implicit in function implementations.

The poor documentation of kernel API functions reduces the chances for developers to produce bug-free kernel-level service code.

4.1.4 Large API

The last specificity of the Linux kernel is that its API includes thousands of functions that cannot be thoroughly understood by a single developer. For example, Linux kernel 2.6.28 released in December 2008 came with about 11,000 exported functions[1]. Our study of the evolution of the kernel API in Section 2.4.2.1 (p. 13) has revealed that the number of Linux kernel API functions has been steadily increasing. For instance, Linux 2.6.37 exports 224 more API functions than 2.6.36 which was released 3 months earlier.

Support: The formulated specificities of the kernel API introduce programming and debugging challenges for kernel-level service developers. To support development activities, it is necessary to provide a tool-based approach for analyzing the kernel API to expose the prevalent safety holes. For the approach to be realistic, it should account for the variety of faults that developers may come across during kernel-level service development. It should also be automated to ease the investigation of the numerous API functions.

4.2 Extra debugging information

OS code does not include sufficient information for tagging execution paths or for describing critical behaviors. In case of crash, the runtime logs thus lack useful information for diagnosing OS failures [YMX+10, YZP+].

Adding information however remains a challenge since the nature of the information needed by developers as well as the adequate location to insert this information are undefined. Furthermore, simply enriching existing log messages cannot solve the problem of missing debugging information since logs are not systematically inserted in many places where they might be necessary.

Support: Debugging will undoubtly benefit from more exploitable information that can bring newly created messages in sensitives execution places. It is thus necessary to devise an approach to identify such places, and for creating the most appropriate messages that could help developers more readily debug kernel-level services.

4.3 Low-overhead dynamic monitoring

Static analysis helps to uncover bugs in driver code. However, as discussed in the study of related work, the existing approaches yields a significant number of false positives which deter kernel developers. Dynamic analysis is therefore relevant to accurately monitor the behavior of kernel-level service code.

Unfortunately, execution efficiency is a key issue in kernel programming. Thus, introducing extra instructions can dramatically degrade the performance of the kernel which in turn will noticeably

[1] Exported functions are kernel API functions that are available to external modules

affect user applications. Even in the early stages of driver development, outside production settings, a minimum degree of efficiency is required.

Support: The performance requirements in kernel programming constrain the use of dynamic monitoring. Thus, to support this approach of debugging, it is necessary to identify key points in kernel code where to interpose runtime checks. These checks and the possible logging instructions must be designed and implemented to only leave a reasonable computation footprint.

4.4 Reliable crash information retrieval

Monolithic OS kernels are notably hard to debug, because when they crash the whole system dies. It is possible to debug a kernel remotely or via a virtual machine, but the former is inconvenient and some kinds of code, such as drivers, typically does not run in a virtual machine, instead the virtualization process uses the native versions. Consequently, initial debugging tends to rely on crash reports, that contain some information about the place where the crash occurred as well as a backtrace of the called functions. Using this information in debugging raises two issues: 1) the reliability of the provided information, and 2) the relevance of the provided information to the actual fault. Debugging kernel hangs raises further issues.

To explicit those issues we consider a bug due to a misunderstanding of the idiosyncratic return value of a kernel API function, namely `open_bdev_exclusive`. Figure 4.1(a) shows an excerpt of the definition of the kernel exported function `open_bdev_exclusive`, which returns a value constructed using the kernel function `ERR_PTR` when an error is detected. Dereferencing such a value will crash the kernel. Thus, this return statement also represents a safety hole. In Linux 2.6.32, in the file *fs/btrfs/volumes.c*, the function `btrfs_init_new_device` called `open_bdev_exclusive` and compared the result to `NULL` before dereferencing the value. This test, however, does not prevent a kernel crash, because an `ERR_PTR` value is different from `NULL`. Yet, many developers still consider `NULL` as the unique error return value. Figure 4.1(b) shows a patch fixing the fault.

```
1  struct block_device *open_bdev_exclusive(
2          const char *path, fmode_t mode, void *holder)
3  {
4      ...
5      return ERR_PTR(error);
6  }
```

a) Excerpt of the definition of `open_bdev_exclusive`

```
1  commit 7f59203abeaf18bf3497b308891f95a4489810ad
2      bdev = open_bdev_exclusive(...);
3  -   if (!bdev) return -EIO;
4  +   if (IS_ERR(bdev)) return PTR_ERR(bdev);
```

b) Excerpt of the bug fix patch

Figure 4.1: Bug fix of error handling code

Reliability of kernel oops reports Linux kernel backtraces suffer from the problem of *stale pointers*, *i.e.*, addresses within functions that have actually already returned. To illustrate this problem, we

consider a crash occurring in the function `btrfs_init_new_device` previously shown in Figure 4.1. The crash occurred because the kernel exported function `open_bdev_exclusive` returns an `ERR_PTR` value in case of an error, while `btrfs_init_new_device` expects that the value will be `NULL`. This caused a subsequent invalid pointer dereference.

To replay the crash, we installed a version of the `btrfs` module from just before the application of the patch. To cause `open_bdev_exclusive` to fail we first create and mount a `btrfs` volume and then attempt to add to this volume a new device that is not yet created. This operation is handled by the `btrfs_ioctl_add_dev` ioctl which calls `btrfs_init_new_device` with the device path as an argument. This path value is then passed to `open_bdev_exclusive` which fails to locate the device and returns an `ERR_PTR` value.

Figure 4.2 shows an extract of the resulting oops report. Line 1 shows that the crash is due to an attempt to access an invalid memory address. Line 5 shows that the faulty operation occurred in the function `btrfs_init_new_device` a priori during a call to `btrfs_ioctl_add_dev` (line 8). Source files and line numbers can be obtained by applying the standard debugger `gdb` to the compiled module and to the compiled kernel.

```
[847.353202] BUG: unable to handle kernel paging request at ffffffee
2 [847.353205] IP: [<fbc722d9>] btrfs_init_new_device+0xcf/0x5c5 [btrfs]
[847.353229] *pdpt = 00000000007ee001 *pde = 00000000007ff067
4 [847.353233] Oops: 0000 [#1] ...
[847.353291] EIP is at btrfs_init_new_device+0xcf/0x5c5 [btrfs] ...
6 [847.353298] Process btrfs-vol (pid: 3699, ...
[847.353312] Call Trace:
8 [847.353327] [<fbc7b84e>] ? btrfs_ioctl_add_dev+0x33/0x74 [btrfs]
[847.353334] [<c01c52a8>] ? memdup_user+0x38/0x70 ...
10 [847.353451] ---[ end trace 69edaf4b4d3762ce ]---
```

Figure 4.2: Oops report following a `btrfs` `ERR_PTR` pointer dereference crash.

This backtrace contains possibly stale pointers, as indicated by the `?` symbol on lines 8 and 9. While `btrfs_ioctl_add_dev` really does call `btrfs_init_new_device`, this is not the case of `memdup_user`. Indeed, it is quite unlikely that a very generic function like `memdup_user` would call a very specific file-system related function like `btrfs_ioctl_add_dev`. These spurious call stack entries seem to come from the use of function pointers, and unfortunately functions pointers are used quite heavily in the kernel. Since it cannot be known a priori whether a function annotated with `?` is really stale, the service developer has to find and study the definitions of all of the functions at the top of the backtrace, until finding the reason for the crash, including the definitions of functions that may be completely unrelated to the problem. A goal of the kernel debugger kdb,[2] which was merged into the mainline in Linux 2.6.35, was to improve the quality of backtraces. Nevertheless, backtrace quality remains an issue.[3]

Relevance of kernel oops reports A kernel oops backtrace contains only the instruction causing the crash and the sequence of function calls considered to be on the stack. The actual reason for a crash, however, may occur in previously executed code that is not represented. For the fault shown in Figure 4.1, the oops report mentions a dereference of the variable *bdev* in the function `btrfs_init_new_device`, but the real source of the problem is at the initialization of *bdev*, to the result of calling `open_bdev_exclusive`. This call has returned and thus no longer appears on the stack.

[2] https://kgdb.wiki.kernel.org/

[3] https://lkml.org/lkml/2012/2/10/129

Such situations make debugging more difficult as the developer must thoroughly consult kernel and service source code to localize important initialization code sites.

Kernel hangs By default, the Linux kernel gives no feedback in the case of a kernel hang. It can, however, be configured to panic when it detects no progress over a certain period of time. When the hang is due to an infinite loop, the backtrace resulting from the panic can occur anywhere within this loop; the point of the panic may thus have no relation to the actual source of the problem.

Support: While backtraces remain a valuable tool for isolating the functions in the call stack, it contain noise that may hide useful information on the instruction at which the fault manifested itself, and it does not necessarily provide a hint to the reason why, at the point the crash, the system failed. Kernel debugging tasks could therefore benefit from tools that more readily and more accurately trace the *origin* of an error and its *cause*.

4.5 Summary

In this chapter, we have explored the different points of kernel debugging that could be improved to offer an enhanced support to kernel-level service developers. First we have shown that kernel code can be large and complex and that kernel APIs require much attention in their usage to avoid runtime errors that appear because developers failed to take into account implicit usage preconditions. Second, we explored the need for creating new log messages in the execution paths of service code so that, when a crash occurs, developers may benefit from more exploitable information. Third, we have re-stated the performance constraints of kernel debugging which imposes a low overhead to runtime monitoring instructions. Finally, we have discussed the current practices of kernel debugging and shown that the kernel backtraces, main available information after a crash, are not rigorously reliable and are not always relevant to pinpoint the cause of a crash.

The investigation on the challenges of debugging tasks in kernel-level service development has driven the design and implementation of the Diagnosys approach [BRLM12]. In this approach, we have chosen to focus our interest in the interactions between a kernel-level service and the core kernel. To study the potential faults that may appear in this interaction we first analyze the entry points, i.e., the kernel APIs, to identify the safety holes that they may contain. To this end, we rely on an existing rule-based analysis tool, namely Coccinelle, that could be used for specifying safety hole criteria. To conclude this *background* part of our thesis, we introduce Coccinelle in the next chapter.

Chapter 5

Static Program Analysis

"First law: the pesticide paradox. Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffective"

Boris Beizer

Contents

To analyze the behavior of computer programs, developers often resort to automated program analysis tools. These are generally applied to check program correctness or to perform program optimization. Some of the tools look at the code, either at the form of text, or in the form of a syntax tree, to determine certain properties of the code such as the interdependency between software modules: such tools are said to perform static analysis. Other tools look at the flow of execution and can be useful for getting the flow of arguments, or determining the time spent in certain parts of the program: these tasks are generally performed by dynamic analysis tools.

Dynamic program analysis is performed by executing programs on a processor. Unfortunately, for this approach to be effective, the target program must be executed with sufficient test inputs to cover all possible behaviors. In the case of an OS, such as the Linux kernel, the variety of execution contexts, of available hardware, hence of device drivers, lead to an important number of configuration possibilities. For example, the Linux kernel, with about 10,000 configurable features [TLSSP11], is even growing rapidly making exhaustive dynamic analysis an impossible endeavor.

Static program analysis, on the other hand, is performed without actually executing the target program. Thus, the analysis is generally performed on the available source code, although it can also be performed on some forms of program object code. In the scope of this thesis, we resort to static analysis techniques to thoroughly study the implementation of kernel APIs. We first explore in Section 5.1 some generalities on static analysis to introduce different notions that will be referred to

in the rest of this document. In Section 5.2, we introduce Coccinelle, the static analysis tool that we have used in the implementation of Diagnosys.

5.1 Generalities on static analysis

Static analysis tools have gained popularity for safeguarding against the most common causes of errors in software by focusing on automatic bug finding. The bug finding tool warns about potential bugs that the human confirms and then corrects. We discuss in the following the approaches of finding bugs with static analysis to highlight their challenges (Section 5.1.1). We also present the possible outcomes of static analysis that justify why bugs still remain to be found in the code of software systems (Section 5.1.3).

5.1.1 Bug finding

To find bugs in software, developers have often relied in various techniques, including software testing, code inspection, model checking, theorem proving, static program analysis and combinations thereof. Static analysis however presents many practical advantages that make it widely used by researchers and practitioners alike.

Software testing consists in running the program with a set of inputs and checking whether it behaves correctly. This approach however presents a number of limitations. First, test scaffolding is time-consuming to create, which often leads to insufficient tests. Second, not all bugs are easily caught during testing. For example, threading bugs can be very hard to reproduce. Third, other bugs are actually difficult to test because they do not necessarily depend on the input tests. For example, error handling code is difficult to exercise.

Code inspection refers to the process of manually examining the source code to look for bugs. The main limitations of this approach is that it is labor intensive and subjective. Indeed, humans are known to have blind spots when reading source code [AHM+08]. Thus, the source code might appear to be correct when actually it is not.

Model checking computes the runtime states of the program without running the program and uses these states to check whether some property holds [Mer01]. Because of undecidability, the model checking approach may fail to prove or disprove a given property. Furthermore, model checking is a far lesser reachable technique for common developers and software designers who must write or provide tools for inferring checking specifications.

The idea behind static analysis approaches is to automate code inspection by using a program to analyze other programs to find bugs. This analysis is performed on program statements, on the general control flow, on function calls, etc., and offers the advantages of being relatively objective. In addition, the automation enables the analysis of many more program behaviors (different control flow paths).

Using static analysis for bug finding has now become an extremely active research area. Some of the more well known static program analysis tools include PREfix [BPS00], SLAM [BMMR01], ESP [DLS02], ESC [FLL+02], MOPS [CW02], and Coccinelle [PLHM08]. The design of these tools however have generally favored one of two important properties, namely scalability and usability.

Scalability Engler *et al.* [ECH+01], Li *et al.* [LZ05], and Padioleau *et al.* [PLM06] have proposed approaches that specifically aimed at being highly scalable. Building on techniques such as model checking, statics and data mining, the proposed approaches were successfully applied to software of millions of lines of code including the Linux kernel. Unfortunately, the limited interaction between

the user and the analysis strategies complicate the use of such approaches. Indeed, these static analysis tools use complex heuristics over which a user has little control [LZ05].

Usability Static program analysis tools rely on user specifications of what is correct or incorrect code. As it is difficult for tool developers and tool users to anticipate specifications for all possible bugs, researchers have devised various strategies for inferring specifications. To improve usability of static bug finding tools, automata-based languages have been proposed for describing code patterns that constitute bugs [ECCH00]. The proposed specifications however remained difficult to relate to the code structure, making it difficult for a user to understand why a code pattern is considered to be part of bug, or why it is overlooked by the analysis tool. Padioleau *et al.* have proposed to write specifications using a language that is very close to C code, the Linux kernel language, in order to ease specification development [PLHM08].

5.1.2 API usage protocols

In earlier sections of this document, we have discussed at length the invalid usage of API functions and explained how it is now one common bug in the Linux kernel. Static analysis tools have been used to identify those API functions, the protocol for their correct usage as well as any code that violates these protocols. Engler *et al.*, building on their experience with their static analysis tool, have identified unspecified or implicit protocols as a "major obstacle" to finding bugs in real systems [ECH+01]. Tan *et al.* have proposed AutoISES [TZM+08] which uses data mining to search for protocols involving access control primitives. In this approach, the user provides the list of access control primitives and the data mining process searches for structure field accesses that these primitives should protect. The WYSIWIB declarative approach [LBP+09] is based on the Coccinelle control-flow based search engine. In this setting, the programmer is able to express bug search specifications, called *semantic matches*, using a syntax that is close to that of ordinary C code.

5.1.3 Analysis failures

A major limitation of static analysis is that, in general, it can never be precise. Instead, it approximates the behavior of the target program either towards completeness (never miss a bug) or towards soundness (never report a false alarm). Balancing between those ends is a key property of current approaches which present different rates of false positives and false negatives.

False positive False positive errors, also know as false alarms or type I errors, are analysis results indicating that the tool has found an issue in a program point where there is really no such an issue. This happens as static analysis cannot always determine whether an apparent execution path is possible or combinations of variable values are possible, leading it to sometimes report potential errors that are not actually possible.

Bug finding tools can be designed towards *completeness*. In this case, the analysis is performed to always overestimate possible program behaviors, so as to never miss a bug. However, it might report some false positive bugs. The analysis may thus report too many false positives that will hide the real bugs which will not be noticed during manual checking. A trivial example of analysis that aims for completeness is the case where the tool reports a bug at every point in the program.

We illustrate, with the code example in Figure 5.1, how a false positive can occur. In this example, static analysis calculates that the set of possible values for x are {6,3} and the set of possible values for y are {3,0}. Static analysis does not know that the combination of values where x and y are both

equal to 3 is impossible, so it issues a false positive error about a possible divide-by-zero error on line 4.

```
1  #include <stdio.h>
2
3  double carelessOp(int x, int y){
4      return 1/ (x−y);
5  }
6
7  int main(int argc, char **argv) {
8      int x,y;
9      if (argc > 3) {
10         x = 6;
11         y = 3;
12     } else {
13         x = 3;
14         y = 0;
15     }
16     printf("%f\n", carelessOp(x,y));
17     return 0;
18 }
```

Figure 5.1: Example of code highlighting "Division by 0" warnings

False negative False negative errors, also known as type II errors, are real bugs that are missed by the static analysis.

Bug finding tools can be designed towards *soundness*. In this case, the analysis is performed to always underestimate possible program behaviors so as to never report a false positive bugs. However, it might miss some real bugs. The analysis may thus report fewer bugs than the ones actually contained in the target program. A trivial example of analysis that aims for soundness is the case where the tool does not report any bugs when analyzing any program.

Table 5.1 summarizes the different possibilities of analysis outcomes at any point of the program.

	Tool **did** find the bug	Tool **did not** find the bug
A bug exists	OK	*False negative*
A bug does not exist	*False positive*	OK

Table 5.1: Combinations between analysis results and reality

There are however practical issues involved with the use of static analysis tools. Given a program with 100 bugs, a static analysis aiming for completeness may find all those 100 bugs while issuing 1,000,000 bug reports. Another analysis aiming for soundness would then find 25 bugs while issuing only 50 bug reports. Many approaches work with the assumption that using a bug finding tool must be a productive use of developer's time and thus aim at fewer false positives. In critical settings, however, missing bugs can be damaging.

Pursuing the right Trade-off

Finding the right trade-off between the requirement for completeness to ensure execution safety of programs and the need for soundness to ease debugging tasks is essential. Thus, a static analysis for

bug finding processes does not need to be consistent in its approximations: current approaches are neither complete nor sound. Current approaches allow the analysis to be flexible in its estimation of *likely* program behaviors, to allow it to be, in general, more precise in the findings.

Nonetheless, it is important to allow static analysis tool users to have more control on the cursor between completeness and soundness. Most static analysis approaches, which aim at being scalable, fail to include this usability requirement. Coccinelle, thanks to its specification language, allows to write specifications, called *semantic matches*, that can be easily tailored so as to eliminate false positives or catch more potential bugs.

5.2 Coccinelle

Coccinelle was originally designed to document and automate *collateral evolutions* in the Linux kernel source code [PLHM08]. It is now used in various code bases as a tool performing control-flow-based program searches and transformations in C code [BDH+09]. Coccinelle is built on an approach where the user guides the inference process using patterns of code that reflect the user's understanding of the conventions and design of the target software system [LBP+09]. Static analysis by Coccinelle is specified using control-flow sensitive concrete syntax matching rules. Coccinelle provides a language, SmPL[1], for specifying search and transformations. It also includes a transformation engine for performing the specified searches and transformations.

In this thesis, we use Coccinelle for writing only *semantic matches* which are used for code searching. Program transformations which are used for bug fixes are not discussed in this document. We present SmPL in terms of simple semantic matches for detecting bad usage protocols of API functions. We furthermore discuss how inter-procedural analysis, which is not integrated into Coccinelle, can be achieved.

5.2.1 Samples of SmPL

SmPL specifications are based on pattern matching, but can contain OCaml or Python code, to be able to perform arbitrary computations. In this section, we describe in details semantic match writing and some of SmPL features that are leveraged in the scope of this thesis work.

The semantic match of Figure 5.2 detects cases where the implementation of a function dereferences unsafely the pointer value of a parameter of this function. Such a dereference is potentially dangerous in kernel code as if the pointer value is invalid (NULL or Error pointer), then the unsafe dereference will crash the kernel. The semantic match consists of two rules: the first (lines 1-15) is named unsafe_dereference and is written in SmPL, and the second (lines 20-30) is written using the SmPL interface to Ocaml. Each rule begins with the declaration of a collection of metavariables and then follows with either a C-code-like pattern specifying a match in the case of a SmPL rule or an ordinary Ocaml code.

The rule unsafe_dereference defines eight metavariables (lines 2-6): *T* (type) which represents any data type, *p*, which represents an arbitrary position in the source program, *new_val*, which represents an arbitrary expression, *fn* (function name), *param* (parameter name) and *fld* (data structure field name), which represents arbitrary identifiers, and *params_prec* (preceding parameters names) and *n* (number of parameters), which represents arbitrary list of parameters and their number. Metavariables are bounded by matching the code pattern against the C source code. For example, the pattern fragment on line 8 will cause *fn* to be bounded to the name of a function in its definition, and cause

[1] Semantic Patch Language

```
1  @unsafe_dereference exists@
2  type T;
3  position p;
4  expression new_val;
5  identifier fn, param, fld;
6  parameter list [n] params_prec;
7  @@
8  fn(params_prec, T *param, ...){
9  ... when != param = new_val
10     when != param == NULL
11     when != param != NULL
12     when != IS_ERR(param)
13     param->fld@p
14 ... when any
15 }
```

```
16 // Ocaml code for printing information on
17 // the unsafe dereference detected with the
18 // semantic match rule <unsafe_dereference>
19 // Print function, filename and linenumber
20 @script:ocaml@
21 fn << unsafe_dereference.fn;
22 t << unsafe_dereference.T;
23 n << unsafe_dereference.number;
24 p << unsafe_dereference.p;
25 @@
26 let p_ = List.hd p in
27 let file = p_.Coccilib.file in
28 let line = p_.Coccilib.line in
29 Printf.printf "FUNCTION [%s] - PARAM: %s (%i)
30     FILE: %s - LINE:%i\n" fn t n file line
```

Figure 5.2: A semantic match to search and report unsafe dereferences of function parameters

param to be bounded to any pointer parameter name and *params_prec* to the list of parameters that precede it. The notation @*p* binds the position variable *p* to information about the position of the match of the preceding token. Once bounded, a metavariable must maintain the same value within the current control-flow path; thus, for example, the occurrences of *param* on lines 8-13 must all match the same expression. The code pattern (lines 8-15) then consists of essentially C code, mixed with a few operators to raise the level of abstraction so that a single semantic match can apply to many code sites.

Sequences The main abstraction operator provided by SmPL is '`...`', representing a sequence of terms. In line 8,'`...`' represents the remaining parameters of a function after a given parameter is matched; in line 9, '`...`' represents the sequence of statements reachable from the begin of the definition of a function along any control-flow path. By default[2], such a sequence is quantified over all paths (e.g., over all branches of a conditional), but the annotation `exists` next to the rule name indicates that for this rule, there need be only one. It is also possible to restrict the kinds of sequences that '`...`' can match using the keyword `when`. Lines 9-12 use `when` to indicate that there should be no reassignment of param nor any check on the validity of the *param* pointer value before reaching the dereference that consists in accessing a field *fld* in the corresponding data structure.

Figure 5.3 shows an example of C function definition code where a parameter is unsafely dereference in one possible execution path. A SmPL rule only applies when it matches the code completely. The rule unsafe_dereference matches the parameter of type *struct person* * on line 1 and the dereference on line 6 as it exists a control-flow path where the validity of *pers* is not checked. The metavariable *fn* is bound to the identifier *get_age*, and *param* is bound to *pers*. The metavariable *p* is bound to various information about the position of the dereference, such as the file name, line number, and so on.

Inheritance A script rule does not match against the source code, but instead *inherits* metavariables from other rules and performs some processing on their values [Stu08]. We use Ocaml rules in this work to print information on matched code patterns. In the example of Figure 5.2, the Ocaml rule inherits, among other metavariables, the identifier metavariable *fn* (line 21) from the rule unsafe_dereference representing the name of the function matched, and the position metavariable

[2]This default behavior can also be explicitly stated using the `forall` annotation

```
1  int get_age(int alive, struct person *pers, char *context){
2      int age=0;
3      if (alive == 1 && pers != NULL)
4          age=pers->age_death - pers->age;
5      else
6          age=pers->age
7      return age;
8  }
```

Figure 5.3: Example of C function code including an unsafe dereference

p (line 24) from the rule unsafe_dereference holding the position of the unsafe dereference. The Ocaml rule prints information on this dereference, including the function name, the relevant parameter number and its number, the file name and the line number on which the dereference occurred.

Disjunctions and Nests A semantic match can consist of multiple rules, each of which can inherit metavariables from any previous ones. A given rule is applied once for each possible set of values of the inherited metavariables. Besides the '. . .', SmPL provides disjunctions, $(pat_1 \mid ... \mid pat_n)$, and nests, $< ... \quad pat \quad ... >$. A disjunction matches any of the patterns pat_1 through pat_n and is used in the semantic match of Figure 5.4 to consider the two kinds of pointers representing error values that can be returned by a function.

```
1  @return_error exists@
2  position p; identifier virtual.fn;
3  expression *E; expression E0;
4  @@
5  fn (...) {
6  ... when any
7  ( return@p NULL; | return@p ERR_PTR (...); )
8  }
```

Figure 5.4: A semantic match to report all functions that may return an invalid pointer value

A nest $< ... \quad pat \quad ... >$ in SmPL matches a sequence of terms, like '. . .'. However, additionally, it can match **zero or more** occurrences of pat within the matched sequence. Another form of nest exists for matching **one or more** occurrences of pat. By analogy to the $+$ operator of regular expressions, this form is denoted $< +... \quad pat \quad ...+ >$.

We have presented these features of Coccinelle to provide a hint of the power that it gives to developers looking to perform a static analysis where they need much control on the writing of the specifications and of how the analysis will be performed across possible control-flow paths. Furthermore, the SmPL language allows a user to easily interject his understanding of the code structure, as well as a project coding conventions, into the bug finding process. This flexibility that SmPL offers to constrain the searches more or less loosely along the control-flow paths, lets the user have control on the potential false positives that a static analysis will yield. The Coccinelle tool has been used in various works to automate collateral evolutions [PLHM08], find and document bugs [Stu08, LBP+09, PLM10, LLH+10], support code transformations [Bis10], etc. In this thesis we leverage the capabilities of Coccinelle to search, categorize and document safety holes in kernel APIs.

5.2.2 Interprocedural analysis

A major caveat in the implementation of Coccinelle is that it does not support interprocedural analysis. However, this kind of analysis, which consists of gathering information across multiple procedures of a program, is important to identify bugs and API usage protocols. Indeed, by extending the scope of control-flow analysis across procedure boundaries, interprocedural analysis accounts for all issues that may span across multiple function definitions or across the entire program. To this end, the analysis incorporates the effects of procedure calls in the calling contexts, and the effects of calling contexts in the called procedures. There is therefore a need to find a workaround to perform thorough static analysis (intra and interprocedurally), while benefiting from the power and flexibility of SmPL.

Since version 0.2.5, Coccinelle has supported iteration of a set of rules over a set of files, which can be used to write specifications for an interprocedural analysis. In this case, the tool user is responsible for integrating the need for interprocedural analysis by specifying which rules must be re-used in iterations over a set of files over which they have complete control. Figure 5.5 shows a semantic match that extends the semantic match of Figure 5.2 by searching for functions that, instead of dereferencing unsafely their pointer parameters, forward them to internal routines that may dereference them unsafely. The rule `parameter_forward` matches (line 13) the name of the internal function and the number of the parameter that is forwarded. The Ocaml script rule (lines 17-32) is then in charge of scheduling this function for a later analysis by calling a function implemented by the user to handle such interprocedural analysis (line 32). A *virtual* identifier (line 20), through which users can specialize the analysis, is used to contain the source code directory where the interprocedural analysis must search for the implementation of the internal function.

Figure 5.6 illustrates how Coccinelle allows iterations to be systematically performed. The Ocaml `initialize` rule contains code that is loaded once before any matching is performed on any SmPL rule for any program source file. In this function, we define two functions that will be called by the different rules. The first function, *interprocedural_analysis* (lines 5-17), is the one that actually registers the iteration to be performed after setting a few values for the Coccinelle search engine. In the second function, *add_for_iteration* (lines 19-24), which is called after a match is found (Figure 5.5), the engine continues with the interprocedural analysis and records the iteration level to allow the program to terminate in large systems such as Linux, where the existence of many functions with the same name can make the analysis run indefinitely.

Coccinelle however has limitations with regards to the lack of data flow analysis. Furthermore, because program analyses are necessarily approximate, the Coccinelle user should understand the limitations of the tool, and iteratively refine his semantic matches to improve the results so as to get the fewer false positives while identifying a large instances of bugs.

5.3 Summary

In this chapter, we have explored the power and challenges of static program analysis, and define a few notions, including false positives and false negatives, that will be regularly referred to in the remainder of this document. Thus, in Section 5.1, we discuss some generalities about static analysis and how it is often used to find bugs in programs or to identify API usage protocols. In section 5.2, we introduce the Coccinelle static analysis tool and the SmPL language for specifying searches. A large part of the implementation work of this thesis is built around the Coccinelle tool, making it necessary to discuss its features. We also discuss how interprocedural analysis can be achieved, to work around a Coccinelle limitation that has been pointed in previous works [LBP+09, LBP+12, PLM10].

```
@parameter_forward exists@
position p;
expression E;
identifier fn, param, fn_in;
expression list [f_number] args_prec;
parameter list [h_number] params_prec;
@@
fn(params_prec, T *param, ...){
... when != param = E
    when != param == NULL
    when != param != NULL
    when != IS_ERR(param)
    fn_in@p(args_prec, param, ...)
... when any
}

@script:ocaml depends on parameter_forward@
p << parameter_forward.p;
fn << parameter_forward.fn;
src_dir << virtual.src_dir;
fn_list << virtual.fn_list;
fn_in << parameter_forward.fn_in;
n_fn << parameter_forward.h_number;
n_fn_n << parameter_forward.f_number;
@@
let p_ = List.hd p in
let arg_number = n_fn + 1 in
let param_number = n_fn_n + 1 in
let hazard_file = p_.Coccilib.file in
let hazard_line = p_.Coccilib.line in
let fn_list = fn_list ^ '-_-' ^ fn in
add_for_iteration src_dir fn fn_in fn_list param_number arg_number hazard_file hazard_line
```

Figure 5.5: Semantic match taking into account interprocedural analysis

```
@initialize:ocaml@
open Str
open Printf

let interprocedural_analysis src_dir fn fn_internal all_fn_list param_number arg_number filename linenumber =
    let it = new iteration() in
    it#set_files [files];
    it#add_virtual_rule Interprocedural_analysis;
    it#add_virtual_identifier Fn fn_internal;
    it#add_virtual_identifier Src_dir src_dir;
    it#add_virtual_identifier Fn_list all_fn_list;
    it#add_virtual_identifier Hazard_file filename;
    it#add_virtual_identifier Hazard_line (string_of_int linenumber);
    it#add_virtual_identifier Param_number (string_of_int param_number);
    it#add_virtual_identifier Fn_arg_number (string_of_int arg_number);
    it#register()
;;

let add_for_iteration src_dir fn fn_internal all_fn_list param_nbr arg_nbr filename linenumber =
    let all_fns = Str.split (Str.regexp_string "-_-") all_fn_list in
    // (* Here, we have chosen not to go deeper than 5 internal functions *)
    if (List.length fns < 6) then
        interprocedural_analysis src_dir fn fn_internal all_fn_list param_number arg_number filename linenumber
;;

```

Figure 5.6: SmPL-based Ocaml script for integrating support for interprocedural analysis

Chapter 6

Thesis Steps

"When it is obvious that the goals cannot be reached,
don't adjust the goals, adjust the steps"
Confucius

Contents

In Chapter 2 (p. 7), we have discussed the challenges for developing kernel-level services in the context of monolithic operating systems, in particular with regards to the use of kernel APIs. We have then explored in Chapter 3 (p. 17) the state-of-the-art approaches for helping developers find and address bugs in system software, and shown the insufficiencies of the different techniques. We have further enumerated in Chapter 4 (p. 25) different points that must be addressed to provide improved support to kernel-level service developers. Among those points are the static analysis of program code which usually yields too many false positives. To account for the necessity for the user to have more control over the analysis, this thesis work builds around a more suited tool, Coccinelle, which was presented in Chapter 5 (p. 31).

We introduce in this chapter the general steps that we have taken to propose an approach for addressing some of the problems encountered by kernel-level service developers in early stages of service development. We describe this approach from the identification of the various issues to the evaluation of our solution.

6.1 Problems

In this section, we consider the practical case of developing drivers for Linux, a monolithic kernel, where new devices are being developed every day for both consumer and industrial systems. Indeed, about 200 new drivers have been integrated to the mainline Linux kernel per year over the last 7 years.[1] In this environment, it is important to provide support for the development of new drivers. Because the importance of Linux to device manufacturers is a fairly recent phenomenon, drivers for new devices are often developed by engineers who are not experts in the design of the Linux kernel.

[1] In all the document, we use the term "the Linux kernel" to refer to the complete mainline Linux kernel source tree, including drivers, filesystems, etc.

To allow driver development to proceed as rapidly as possible, such engineers could benefit from the support of a debugging interface to the OS. At a minimum, such an interface should provide a well-defined set of entry points that fail gracefully, while leaving traces, when invoked on invalid inputs or from an invalid execution context, and whose traces provide understandable log messages describing the reason for any failure.

Unfortunately, Linux has many weaknesses in this regard: (1) Linux does not provide a well-defined interface. Instead, the set of exported functions is fluid and their implementation tends to favor efficient execution over robustness to common programming errors. This complicates the initial driver development process, as developers have to identify the functions that are available and then study their definitions, as well as the definitions of any other function they call, to determine how to use them correctly. Furthermore, (2) Linux drivers run in privilege mode and their failure, which impacts the entire kernel, are hard to diagnose because of the reliability and relevance issues of the kernel oops reports (cf. Section 4.4) (p. 28).

The issues enumerated in this section and in previous parts of this document can also be found in other systems, including other monolithic kernels and other software systems that are built based on the plug-in architecture model. The various research work that we have presented in Chapter 3 (p. 17), propose different approaches that have shown their limitations in addressing fully the challenges of kernel software development. It therefore appears that devising a tool that aims at significantly reducing the pains for early-stage driver development is still of interest to the community. We now present the general steps that we have followed to design and implement our approach.

6.2 Approach and steps

To ease and improve debugging tasks in kernel-level service development, we propose an approach that focuses on the interactions between a kernel-level service and the rest of the kernel. We propose a mixed approach with different steps including static analysis of source code, code instrumentation, runtime monitoring of service behavior, and adapted tools for helping developers readily test and recover debugging informations after failures. Figure 8.1 illustrates the general process that we define for supporting developers of kernel-level services.

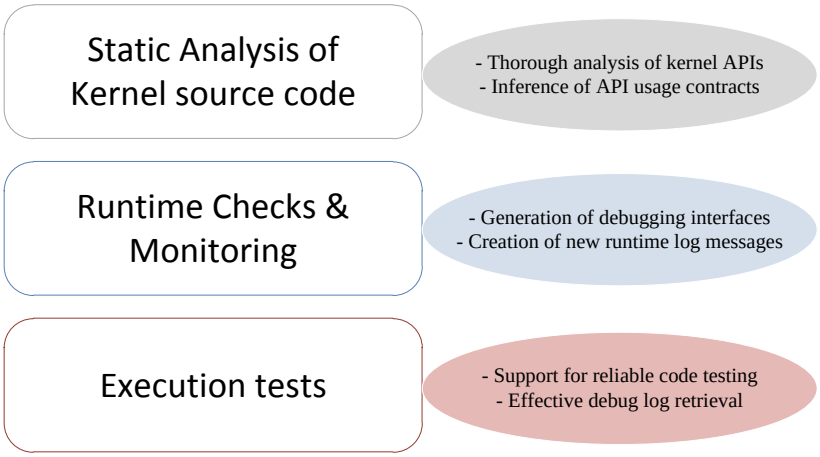

Figure 6.1: Different contributions points for improving testing and debugging tasks in kernel-level service development

The different steps of this thesis, which will be extensively reported in the next part of the document, are as follows:

- The first step consists in building analysis tools for identifying in kernel APIs all important usage contracts that may be overlooked by kernel-level service developers. This supposes an intimate knowledge of the kinds of errors that may arise in the usage of kernel interfaces and the possibility of writing search specifications for categorizing API functions that are prone to such errors. Our work thus starts with a study of bug reports filed for code in the Linux mainline tree. Then, we propose a taxonomy of *safety holes* in the implementation of kernel API functions based on our findings and of previous work on empirical studies on operating system errors [CYC+01, PST+11].

- In the second step, we wish to take into account the categories of safety holes that may be contained in API functions used by a given kernel-level service. To this end, we devise a generation process for producing debugging interfaces that will accompany the service code. In this step we help developers by adding, during execution, extra-debugging information that might be valuable to trace back the cause of a system failure. This step is supported by the literature where regular log messages have been shown to be insufficient for diagnosing failures [YZP+].

- The third step consists of the testing phase of kernel-level services. We implement a runtime support comprised of a crash-resilient logging system allowing all stored log information to remain readily available even after a kernel crash or hang. This step is important to leverage all the information collected in steps 1 and 2, and requires an instrumentation of the kernel memory management, of kernel behaviors when a failure occurs, as well as of the reboot system to avoid information lost.

- The fourth, and last, step concerns the evaluation of our approach and of the implemented tools. We first assess, based on our findings in recent kernel versions, whether safety holes have a significant impact on the quality of kernel-code. Then, we study whether the tools that we have built allow to cover a large set of safety hole-related failures. We furthermore investigate how the log information obtained with our approach compares to the kernel oops report. Finally, we evaluate the performance penalty that our approach introduces in the execution of kernel-level services.

The structure of the remainder of this document follows the steps thus introduced. Before introducing the Diagnosys approach, we report on the study that we have conducted to characterize safety holes (cf. Chapter 7 (p. 47). We detail the Diagnosys approach in Chapter 8 (p. 8 and describe debugging replay scenarios with Diagnosys in Chapter 9 (p. 71). Assessment of the Diagnosys approach is proposed in Chapters 10 (p. 77) and 11 (p. 83). Chapter 12 concludes this document, followed by some annexes on the Diagnosys implementation.

Part II

Proposed approach

Chapter 7

Characterization of Safety Holes

"If debugging is the process of removing software bugs,
then programming must be the process of putting them in."
Edsger Dijkstra

Contents

In kernel programming, we consider a safety hole to be **a fragment of kernel code that can cause the kernel to crash or hang if the code is executed in a context that does not respect some implicit preconditions**.

In this chapter, we report on our study of the characterization of safety holes in kernel code. We have first investigated bug fixes reported in kernel logs to identify bugs that relate to the misuse of API functions. We restrict the set of locations of operations that cause a kernel crash or hang to those that are reachable from an exported function. We describe in Section 7.1 a few examples of such programming errors. We then discuss in Section 7.2 the methodology for characterizing safety holes in kernel APIs and propose a taxonomy.

7.1 Examples of safety holes

The Linux kernel does not define a precise internal API. Thus, in identifying safety holes, we focus on the set of functions that are made available to dynamically loadable kernel modules. Such functions are exported using either *EXPORT_SYMBOL* or *EXPORT_SYMBOL_GPL*. In early stages of kernel code programming, developers resort to dynamically loadable kernel modules as a convenient means

to develop new services. Indeed, they allow the service to be loaded into and removed from a running kernel for testing new service versions. In the remainder of this document, *kernel exported functions* are considered as the API functions.

To understand the challenges posed by safety holes, we consider in this section some typical examples in Linux kernel internal API functions and the problems that they have caused, as reflected by Linux patches. These examples involve unchecked parameters, error handling strategies, critical sections and interrupt management.

7.1.1 API function parameters

An *unchecked parameter* safety hole occurs when the value of a parameter of an exported kernel function can trigger a dereference of an invalid pointer, either the parameter itself or another value derived from it. Since kernel code runs in a single protection domain, dereferencing an invalid pointer will crash the entire system. Yet, many exported kernel functions do not check their parameters, for efficiency reasons, because the in-kernel call sites already ensure that the corresponding arguments are valid pointers. For example, in Linux kernel 2.6.32, over 8,000 parameter values from 7,000 (60%) of API functions are dereferenced without any checks. Nonetheless, service developers can assume that these checks are performed or they simply forget to ensure that arguments to such functions are valid.

In-tree code bugs. In the simplest case, the definition of an exported function directly dereferences an unchecked parameter. Because any crash incurred by such a dereference occurs in the function called directly by the service code, it should be easy for a developer to track down the source of the problem. Figure 7.1(a) shows an excerpt of the definition of the exported function `skb_put`, which dereferences its first argument without first checking its value. Many kernel functions are written in this way, assuming that all arguments are valid. This code represents a safety hole, because the dereference is invalid if the corresponding argument is `NULL`. Such a fault occurred in Linux 2.6.18 in the file *drivers/net/forcedepth.c*. In the function `nv_loopback_test`, `skb_put` is called with its *skb* argument being the result of calling `dev_alloc_skb`, which can be `NULL`. The fix, as implemented by the patch shown in Figure 7.1(b), is to avoid calling `skb_put` in this case. `skb_put` remains unchanged.

```
1  unsigned char *skb_put(struct sk_buff *skb, ...)
2  {    unsigned char *tmp = skb_tail_pointer(skb);
3       SKB_LINEAR_ASSERT(skb);
4       skb->tail += len; ...
5  }
```

a) Excerpt of the definition of `skb_put`

```
1  commit 46798c897e235e71e1e9c46a5e6e9adfffd8b85d
2       tx_skb = dev_alloc_skb(pkt_len);
3  +    if (!tx_skb)
4  +         { ... goto out; }
5       pkt_data = skb_put(tx_skb, pkt_len);
```

b) Excerpt of the bug fix patch

Figure 7.1: Bug fix of the usage of `skb_put`

It may also occur that an exported function forwards an unchecked parameter to another, possibly non-exported, function that then dereferences an unchecked pointer argument. In that case the safety hole is interprocedural and the danger that it poses may be more difficult to spot. For example, the exported function *kmap*, which takes as argument a single pointer-typed value, does not dereference its parameter directly. Instead, *kmap* forwards it via the *PageHighMem* macro (Figure 7.2(a), line 5) to the *page_zone* function, which in turn forwards the pointer, again without ensuring its validity, to the function *page_to_nid*. The function *page_to_nid* then dereferences its parameter in line 5

(Figure 7.2(b)). Because the definitions of the various functions may be dispersed in different kernel files and directories, tracing a dereference bug to its origin can be tedious.

```
1  // arch/x86/mm/highmem_32.c
2  void *kmap(struct page *page)
3  {    might_sleep();
4       if ( !PageHighMem(page)) return page_address(page);
5       return kmap_high(page);
6  }... EXPORT_SYMBOL(kmap);
```

a) The API function forwards a parameter

```
1  // include/linux/mm.h
2  static inline int page_to_nid(struct page *page)
3  {
4       return ( page->flags >> NODES_PGSHIFT)
5                    & NODES_MASK;
6  }
```

b) An internal function dereferences the forwarded parameter

Figure 7.2: The *page_to_nid* internal function unsafely dereferences the parameter value of the exported function *kmap*

Figure 7.3 shows a fragment of in-kernel code that calls *kmap*. In this code, the argument to *kmap* is *page*, which is the unchecked result to a previous call to *read_mapping_page*. *read_mapping_page*, however, can return an error value constructed by the function *ERR_PTR*,[1] and *kmap* does not check for such an argument. Instead, *kmap* dereferences its argument untested, which, on an *ERR_PTR* value would cause a kernel crash. This bug was found and fixed in Linux 2.6.28.

```
1  commit 649f1ee6c705aab644035a7998d7b574193a598a
2  int hfsplus_block_allocate(...) { ...
3       page = read_mapping_page(...);
4  +    if (IS_ERR(page)) { start = size; goto out; }
5       pptr = kmap(page); ...
6  }
```

Figure 7.3: An incorrect use of an exported function and the associated fix patch

Finally, we have observed that a value derived from a parameter of an exported function may determine the return value of some other function (possibly non-exported), and that return value is dereferenced without being checked. In this case, the dereference bug is triggered on a variable that appears to be decorrelated from the parameters of the function. Bugs deriving from unchecked parameter safety holes in this category are thus harder to debug.

The exported function *uart_resume_port* (Figure 7.4(a)) passes an unchecked field of the *uport* parameter to the function *device_find_child* (Figure 7.4(a), line 4). When *device_find_child* receives an invalid pointer as its first argument, it returns an invalid pointer, *NULL*, as the result (Figure 7.4(b), line 4). This result is then dereferenced after the call return in *device_may_wakeup* (Figure 7.4(c), line 2). If this result is *NULL*, the dereference will crash the kernel. In this case, the value that causes the crash is not that of a parameter to an exported function itself, but is derived from the value of one.

7.1.2 Error handling strategies

The Linux kernel is written in C, which, contrary to modern programming languages, does not support exceptions. As a result, a function's return value is the primary indicator of the outcome of its execution. Nevertheless, developers often fail to check or even record return values. The annotation `__must_check`, which expands to gcc's `warn_unused_result` attribute, is used occasionally on Linux function definitions to inform the compiler that the result of the function must be stored,

[1] In the rest of this paper, we refer to a value constructed with *ERR_PTR* as just "*ERR_PTR*"

```
1  // drivers/serial/serial_core.c
2  int uart_resume_port(..., struct uart_port *uport)
3  {    ... // No check of 'uport->dev' here
4       tty_dev = device_find_child(uport->dev, ...);
5       if (... && device_may_wakeup(tty_dev)) ...
6  } ... EXPORT_SYMBOL(uart_resume_port);
```

a) A usage of a data variable is questioned

```
1  // drivers/base/core.c
2  struct device *device_find_child
3       (struct device *parent, ...)
4  {
5    ... if (!parent) return NULL;...
6  }
```

b) The routine fails if the pointer parameter is invalid

```
1  // include/linux/pm_wakeup.h
2  inline int device_may_wakeup
3               (struct device *dev)
4  { return dev->power.can_wakeup
5               && ...
6  }
```

c) Dereferencing a NULL pointer *dev* crashes the kernel

Figure 7.4: A dereference bug will occur in the execution of *uart_resume_port* if its parameter *uport* is not correctly initialized

with the intention that the result will be checked, although gcc does not currently enforce the latter. Nevertheless, even though in *e.g.*, Linux 2.6.32, there are almost 4000 exported functions that return an error code, we have found that in the last 10 Linux versions, only about 100 exported functions in each version are marked as `__must_check`. In addition, exported functions have different conventions of error values that can cause dangerous misinterpretations. For example, while most functions return a negative error code, some return a positive error code. Driver programmers may thus fail to appropriately handle a failure, especially when finding out the type of error code is not obvious. Indeed, exported functions can simply forward the return values of internal API functions requiring iterative analysis to detect other possible return values.

In-tree code bugs. Allocation, initialization and some copy functions may fail. Yet their execution outcome, which is usually essential for further processing, is often left unchecked, leading to bugs that have various impacts on the OS. The *copy_from/to_user* exported functions, which copy a block of data from/to user space, return the number of bytes that could not be copied. A positive return value therefore represents an error. This convention is supposed to be widely understood by kernel developers. Nevertheless, kernel change logs show that in each of the last 10 versions of Linux, bug fix patches have been committed to check the return value of these exported functions properly. Figure 7.5 shows an example of such a fix in recent code from the Linux mainline tree.

```
1  commit d3553a52490dcac54f45083f8fa018e26c22e947
2  static long vhost_net_ioctl(...){
3       ...
4  -    return copy_to_user(featurep, &features, sizeof features);
5  +    if (copy_to_user(featurep, &features, sizeof features))
6  +         return -EFAULT;
7  +    return 0;
8       ...
9  }
```

Figure 7.5: An incorrect use of an exported function and the associated fix patch

As another example, Figure 7.6(a) shows an excerpt of the definition of the kernel exported function `open_bdev_exclusive`, which returns a value constructed using the primitive `ERR_PTR` when an error is detected. Dereferencing such a value will crash the kernel. Thus, this return statement also represents a safety hole. In Linux 2.6.32, in the file *fs/btrfs/volumes.c*, the function `btrfs_init_new_device` calls `open_bdev_exclusive` and compares the result to `NULL` before dereferencing the value. This test, however, does not prevent a kernel crash, because an `ERR_PTR` value is different from `NULL`. Figure 7.6(b) shows a patch fixing the fault.

```
1  // fs/block_dev.c
2  struct block_device *open_bdev_exclusive(
3          const char *path, fmode_t mode, void *holder)
4  {   ...
5      return ERR_PTR(error);
6  }
```

a) Excerpt of the definition of `open_bdev_exclusive`

```
1  commit 7f59203abeaf18bf3497b308891f95a4489810ad
2      bdev = open_bdev_exclusive(...);
3  -   if (!bdev)
4  -       return -EIO;
5  +   if (IS_ERR(bdev))
6  +       return PTR_ERR(bdev);
```

b) Excerpt of the bug fix patch

Figure 7.6: Bug fix of error handling code

7.1.3 Critical sections

While many Linux critical sections both begin and end within the body of a single function, some span function boundaries, implying that they expect to be called within a critical section. When the calling code fails to respect this condition, deadlock may ensue.

In-tree code bug. Between Linux 2.6.32 and 2.6.33, a bug was first introduced and then fixed in the `nouveau drm` driver for nVidia® cards, in which the *ttm_bo_wait* exported function was called by the *nouveau_gem_ioctl_cpu_prep* without holding the *bo* lock. Since *ttm_bo_wait* attempts to release and then re-acquire that lock, as depicted in Figure 7.7(a), a subsequent call to another function, *nouveau_bo_busy*, which acquires the lock, hanged the machine. This bug was fixed by ensuring that the *bo* lock is held when invoking the *ttm_bo_wait* API function (cf. Figure 7.7(b)).

```
1  // drivers/gpu/drm/ttm/ttm_bo.c
2  int ttm_bo_wait(struct ttm_buffer_object *bo, ...){ ...
3      spin_unlock(&bo->lock);
4      driver->sync_obj_unref(&tmp_obj);
5      spin_lock(&bo->lock); ...
6  }
```

a) Excerpt of the definition of `ttm_bo_wait`

```
1  commit f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6
2  if (req->flags & NOUVEAU_GEM_CPU_PREP_NOBLOCK){
3  +   spin_lock(&nvbo->bo.lock);
4      ret = ttm_bo_wait(&nvbo->bo, false, false, no_wait);
5  +   spin_unlock(&nvbo->bo.lock);
6  }
```

b) Excerpt of the bug fix patch

Figure 7.7: Bug fix of critical section code

7.1.4 Enabling and disabling interrupts

In Linux kernel code, the implementation of a critical section often both takes a lock and disables interrupts, to prevent a context switch or other interruption that may increase the critical section's duration. The interrupt state is also changed when kernel code explicitly does or does not want to be notified of external events. While locking is a local property, interrupt state is global to a given processor. This introduces the possibility of two types of safety holes:

i) a safety hole where an exported function unconditionally turns on interrupts, as an external driver may assume that interrupts are turned off for the duration of the call;

ii) a safety hole where the execution of an exported function ends by turning interrupts off, as an external driver may assume that interrupts are turned on after the call.

In-tree code bug. A bug due to the first kind of *interrupt precondition* safety hole has been detected in the implementation of the Reliable Datagram Sockets (RDS) protocol which provides reliable delivery of datagrams over Infiniband and iWARP [BT05]. In the code for this protocol, the exported

function *set_page_dirty* unconditionally turns on interrupts. This behavior implies that *set_page_dirty* should not be called with interrupts turned off. To ensure this property, code has been added to the definition of *set_page_dirty* to deliberately halt the system with a kernel panic when this precondition is not respected. Figure 7.8 shows the excerpt of the fix patch.

```
1  commit 561c7df63e259203515509a7ad075382a42bff0c
2  +    BUG_ON(in_interrupt());
3       set_page_dirty(page);
```

Figure 7.8: Bug-fix taking into account a safety hole related to interrupt management

7.2 Taxonomy of safety holes

The examples revisited in previous sections illustrate (1) how safety holes lead to bugs that reach the mainline kernel and that may remain unnoticed for a long time, and (2) how they involve various types of bugs. We now describe here how, based on this investigation, we were able to define a methodology for recasting a type of code fault as a collection of one or more safety holes. We then use this methodology to enumerate the kinds of safety holes considered in this thesis.

7.2.1 Methodology

As illustrated by the bug fixes in Section 7.1, some fragments of code executed by kernel API functions, while themselves being correct, can provoke kernel crashes or hangs when the kernel is used incorrectly. In the most obvious case, the service code passes the API function some bad arguments or calls it from an inappropriate context (e.g., with a lock held or while interrupts are disabled) causing an execution error in the core kernel: we refer to this kind of safety holes as *entry* safety holes. In other cases, the kernel API function may return a value or from a context that might not meet the expectations of the service code: we refer to this kind of safety holes as *exit* safety holes. Such *exit* safety holes appear when, for example, the API function returns *NULL* rather than a pointer to a valid region of memory, or returns with a lock held or with interrupts turned off. In these cases the crash or hang may happen in the service code, with the API function that caused the problem no longer on the call stack. Figure 7.9 illustrates the two families of *entry* and *exit* safety holes, by representing where, in each case, the execution error occurs. In the figure, the box represents the definition of an API function, while the symbol $\perp$ indicates the point of crash.

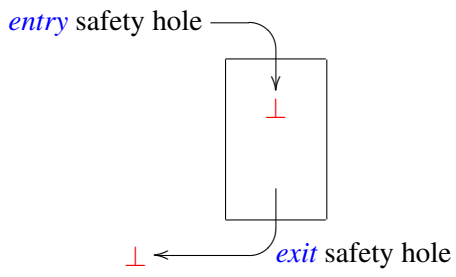

Figure 7.9: Schematisation of the two families of safety holes

More concretely, we observe that any bug type that involves multiple disjoint core fragments can lead to an entry or exit safety hole. We illustrate this observation with a schema in Figure 7.10 from an

example of bug involving a lock/unlock combination for defining a critical section. In this example, the lock and unlock have been written correctly (i.e., the lock is followed by a corresponding unlock). In practice however, a possible bug would be forgetting the unlock, or doing the unlock when the lock has not taken place (e.g., characteristic of a double unlock). Based on the observation that a bug is possible in the interaction between a (or an expected) lock and an (or an expected) unlock, we can split this code inside an exported function in two ways to identify possible exit and entry safety holes. Thus, if a kernel API function ends with a lock (i.e., no unlock), it implements an exit safety hole, because the service code may miss in its implementation the corresponding unlock. Likewise, if the API function begins with an unlock (i.e., no lock), this represents an entry safety hole, because the service may not have previously done what is necessary to take the lock.

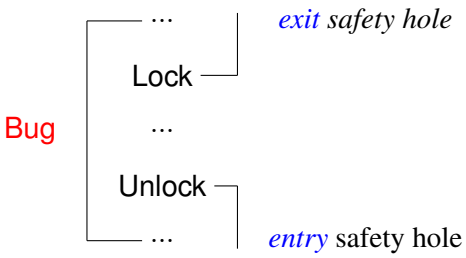

Figure 7.10: Schematisation of safety hole code fragments in a Lock bug

This simple example illustrates the methodology that will be used to enumerate all categories of safety holes in our taxonomy. This taxonomy will then drive the implementation of Diagnosys, our tool for supporting developers of kernel services to readily deal with kernel faults that occur at the boundary between their services and the core kernel.

7.2.2 Taxonomy

The methodology depicted above shows that, broadly, we can consider any bug type involving multiple fragments of code (one of which being inside an API function), and use them to derive definitions of *entry* and *exit* safety holes. To propose our taxonomy we refer to the literature and consider the set of bug types that were categorized by Chou *et al* [CYC+01] and later by Palix *et al.* [PST+11] in empirical studies of faults in Linux code, respectively in 2001 and 2011. Based on this faults, we derive the kinds of safety holes identified.

A safety hole being a fragment of code inside the definition of an API function, it belongs to an execution path. We distinguish between executions paths that are *possibly* followed during a call to the API function, from execution paths that are *certainly* followed. Figure 7.2.2 illustrates a combination of executions paths from an entry point (noted as ●), on call of an API function, to any of the different potential exit points (noted as ■). From this graph, we observe that the execution paths pass through different program points (noted as ○) and are split at some of these points (noted as ◇). A safety hole-related crash point may be located at program points that are reached by all execution paths. We refer to such a safety hole as a *certain* safety hole, indicating that when using the API function, the service code **must** account the presence of this safety hole. Other safety hole crash points however are located at program points that are placed in execution paths that may not be followed by the execution of the API function. Such safety holes are referred to as *possible* safety holes since calling the API function may not involve the execution of the relevant code fragments. The illustration of Figure 7.2.2 describes the common cases of *entry* safety holes indicating that these can be *possible* or *certain*. *Exit*

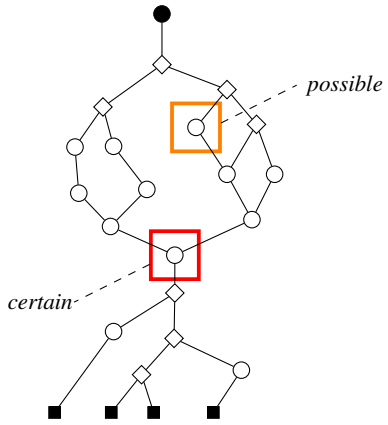

Figure 7.11: Localization of *possible* and *certain* safety holes in the execution paths of an API function

safety holes, on the other hand, are always only *possible* since once the exported function has returned, there is no way of knowing, based on the study of kernel code, what will happen in the execution of the service code.

We now discuss the categorization of safety holes following the categories of faults established through empirical studies. We have already described and presented this categorization of common Faults in Section 3.2 (p. 19). For some of the categories, however, we have not been able to derive safety holes for various reasons that are further developed in the following:

- **NullRef** (*A pointer is dereferenced and then tested for being* `NULL`.) seems meaningless in the context of safety hole search. Indeed, on one hand, if the dereference fails, then the machine has crashed. On the other hand, if the dereference succeeds, this value will not cause the machine to crash.

- **Float** (*Do not use floating point in the kernel.*) is a purely local property. Thus, it is not relevant to the interface between a service code and the kernel exported functions.

- **Real** (*Do not leak memory by updating pointers with potentially* `NULL` *realloc return values.*), in practice, is also a purely local property.

For the remaining categories of faults, we develop in Table 7.1, the different derived kinds of safety holes. For each fault type, we split the possible bug code fragment into two parts and characterize the part that may be located inside the implementation of an API functions. Thus, for each fault type, we provide a summarized definition of the possible entry and exit safety holes that an API function, f, can contain in its definition. The terminology of faults used in Table 7.1 is the same as proposed by Palix *et al.* [PST+11] based on the seminal categorization by Chou *et al.* [CYC+01]. We note in this terminology, that a few categories have been augmented to take into account new kernel functionalities. For example, in kernel 2.6, *ERR_PTR*, along *NULL*, is widely used as an invalid pointer indicating errors. Thus, in the Null and INull categories, we add `ERR_PTR` as an invalid pointer. The same reasoning goes for LockIntr which was added to account for new primitives, such as *spinlock_irq*, that manipulate simultaneously locks and interrupts. For more details, see Appendix A (p. 93).

Block	*To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held.*	
	Entry safety hole	f possibly/certainly calls a blocking function
	Exit safety hole	f returns after disabling interrupts or while holding a lock
Null	*Check potentially* `NULL` *pointers returned from routines.*	
	Entry safety hole	f possibly/certainly dereferences an argument without checking its validity
	Exit safety hole	f returns `NULL`/`ERR_PTR` pointer
Var	*Do not allocate large stack variables (> 1K) on the fixed-size kernel stack.*	
	Entry safety hole	f possibly/certainly allocates an array whose size depends on a parameter
	Exit safety hole	f returns a large value
INull	*Do not make inconsistent assumptions about whether a pointer is* `NULL/ERR_PTR`.	
	Entry safety hole	f possibly/certainly dereferences an argument without checking its validity
	Exit safety hole	f returns `NULL`/`ERR_PTR` pointer
Range	*Always check bounds of array indices and loop bounds derived from user data.*	
	Entry safety hole	f possibly/certainly uses an unchecked parameter to compute an array index
	Exit safety hole	f returns value obtained from user level.
Lock	*Release acquired locks; Do not double-acquire locks*	
LockIntr	Entry safety hole	f possibly/certainly acquires a lock derived from a parameter
	Exit safety hole	f returns without releasing an acquired lock
Intr	*Restore disabled interrupts.*	
LockIntr	Entry safety hole	f possibly/certainly calls a blocking function
	Exit safety hole	f returns with interrupts disabled
Free	*Do not use freed memory.*	
	Entry safety hole	f possibly/certainly dereferences a pointer-typed parameter value
	Exit safety hole	f frees memory derived from a parameter
Param	*Do not dereference user pointers.*	
	Entry safety hole	f possibly/certainly dereferences a pointer-typed parameter
	Exit safety hole	f returns a pointer-typed value obtained from user level
Size	*Allocate enough memory to hold the type for which you are allocating.*	
	Entry safety hole	f possibly/certainly allocates memory of a size depending on a parameter
	Exit safety hole	f returns an integer value

Table 7.1: Recasting Common faults in Linux into safety hole categories. f refers to an API function

7.3 Summary

In this chapter, we have described an investigation of Linux mainline tree code bugs and their fixes to provide an insight of the prevalence of safety holes in kernel code. This investigation has allowed us to devise a methodology for recasting any operating system common fault type into safety hole kinds involving API functions. Based on the proposed methodology, we have defined a taxonomy of safety holes that corresponds to the categories of faults discussed in empirical studies on operating system errors. We refer to this taxonomy in the remainder of this document where we will further detail the search for safety hole instances in kernel code and how to use these results to support kernel-level service development.

After having characterized the safety holes contained in the implementation of kernel API functions, we propose an approach for helping Linux kernel-level service developers to address efficiently the problems that such safety holes may cause during testing and debugging phases of their services.

Chapter 8

Diagnosys

"There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies,
and the other way is to make it so complicated that there are no obvious deficiencies.
The first method is far more difficult."
C.A.R Hoare

Contents

The goal of Diagnosys is to improve the quality of the information available when a crash or hang occurs. To this end, we focus on the various opportunities of crashes and hangs that result from safety holes in kernel API functions. As previously discussed, the motivation for such a focus is threefold: (i) kernel API functions are somewhat an "*unknown territory*" for kernel-level service developers; (ii) API functions are entry points to the kernel code that are not safe; (iii) limiting the focus at the boundary between service code and core kernel code is an efficient way to improve the reliability of kernel code testing.

Figure 8.1 illustrates the steps in using the Diagnosys approach, which involves three phases:

1. Identification of safety holes in kernel API functions and inference of the associated usage preconditions, using the static analysis tool SHAna. This phase is carried out only once by a kernel maintainer, for each new version of the mainline Linux kernel. Indeed, although each

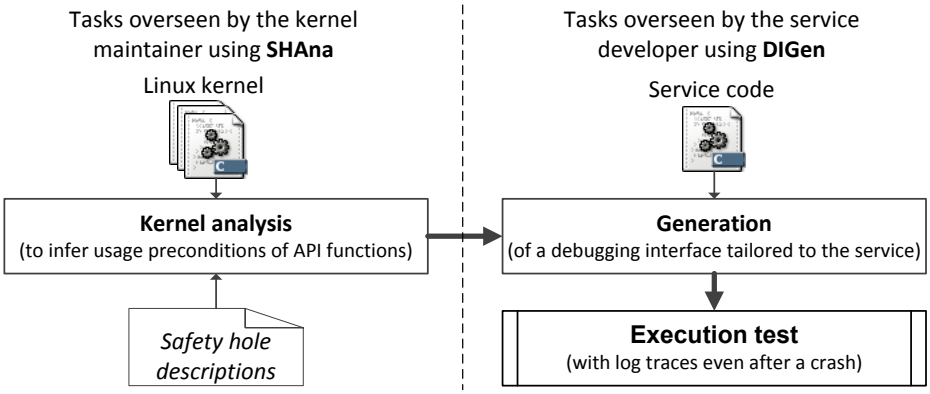

Figure 8.1: The steps in using Diagnosys

Linux distribution may add some specific patches to the Linux kernel, these are unlikely to affect the kernel API. Furthermore, a service that should ultimately be integrated into the mainline kernel must be developed against the API supported by that kernel.

2. Automatic generation of a debugging interface using DIGen based on the inferred preconditions. This phase is carried out by each service developer for each specific service under development.

3. Testing service code with the support of the debugging interface and of the CRELSys crash resilient logging system. This phase is also carried out by each service developer who would like to use Diagnosys.

In this chapter we succinctly describe the design of SHAna, DIGen and CRELSys. Some implementation details are discussed and others are included in the appendix section of this document. We start by discussing the identification of safety holes and the inference of usage preconditions in Section 8.1. Section 8.2 elaborates on the generation process of debugging interfaces. Finally Section 8.3 presents a short overview of the runtime support that is included in Diagnosys for ensuring the preservation of runtime logs.

8.1 SHAna: Safety Hole Analyzer

SHAna is a collection of tools built on top of Coccinelle for identifying safety holes in kernel API functions and inferring their usage preconditions. Figure 8.2 shows the inputs and output of the precondition inference process that is run through SHAna.

Based on the safety hole descriptions, SHAna first searches the kernel code for occurrences of safety holes in the implementations of API functions and then computes the preconditions that must be satisfied for these safety holes to cause a kernel crash or hang. The analysis focuses on unsafe operations that occur in code that is in, or reachable from, an API function. For each such occurrence, a backward analysis amounting to a simple version of Hoare logic [HR00] produces the weakest precondition to be satisfied such that the safety hole may cause a crash. These weakest precondition is computed on entry to the function, for entry safety holes, and on exit from the function, for exit safety holes.

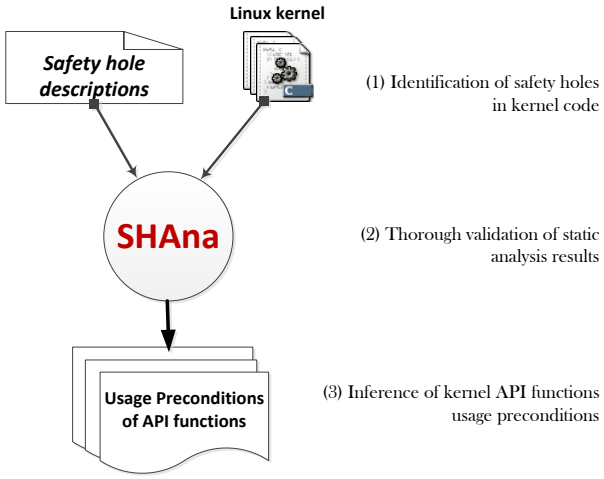

Figure 8.2: The Precondition inference process

8.1.1 Theory of precondition inference

Hoare logic, also known as Floyd-Hoare logic as it was based on an idea of Robert Floyd, is a formal system proposed by Hoare [Hoa69] for reasoning about the correctness of computer programs. In particular this logic introduces the *Hoare triple* to describe how the execution of a program code changes the state of the computation.

A Hoare triple (8.1) consists of two assertions, the precondition P and the postcondition Q, and a program S. The predicate logic states that : *If the execution starts in a state where the precondition P is met, at the end of the program execution the postcondition is realized.* Consider the simple Hoare triple example (8.2) where the precondition $x = y$ is true before the execution of the program. Once the program is executed, the post-condition is true as the value of y has not been changed, and is still equal to the initial value of x. Thus, at the end of execution, we have $x = x_{initial} + 3 \Leftrightarrow x = y + 3$.

$$\{P\}S\{Q\} \tag{8.1}$$

$$\{x = y\}x := x + 3\{x = y + 3\} \tag{8.2}$$

This logic can be used to establish function specifications, i.e., the contract between the implementation of a core API function and the client plug-in (the service code). In this case, the precondition represents the predicate that describes the condition that the API function relies on for correct operation. The postcondition on the other hand describes the condition that the function establishes after correctly running. If some service code calls the API function after fulfilling the function's precondition, the function will execute to completion and when it terminates, the postcondition will be true.

Starting with a post-assertion, i.e., given a postcondition, the weakest precondition $wp(S, Q)$ of S with respect to Q is defined by Equation 8.3 where the program is allowed to be invoked in the most general condition P that would still lead to the realization of the postcondition Q.

$$P = wp(S, Q) \Leftrightarrow \{P\}S\{Q\} \wedge \forall P', \{P'\}S\{Q\} \Rightarrow (P' \Rightarrow P) \tag{8.3}$$

Using a simplified version of this logic we infer the weakest preconditions of API functions in presence of the different kinds of safety holes. For ease of prototyping, we use the program matching

tool Coccinelle [PLHM08] to implement an interprocedural static analyzer that finds the safety holes and constructs the preconditions.

In our search analysis, the postcondition is satisfied when the execution of the API function completes without leading to an execution error for entry safety holes and without returning invalid values or leaving a context that is unsafe for the execution of service code. As previously defined in Section 7.2 (p. 52), a precondition associated with an entry safety hole is classified as :

- *certain*, if satisfaction of the precondition is guaranteed to result in a crash or hang within the execution of the kernel API function
- *possible*, if satisfaction of the precondition may cause a crash or hang on at least one possible execution path

The preconditions of exit safety holes are always classified as *possible*. The result of SHAna is a list mapping each kernel API function identified as containing safety holes to the associated preconditions.

8.1.2 Analysis process

The search analysis for identifying safety holes and inferring their preconditions starts from the definition of an API function, which in our case is a function recognized as one that is declared using *EXPORT_SYMBOL* or *EXPORT_SYMBOL_GPL*. We provide in Table 8.1 a complete summary of the description of the safety holes for all categories of faults. The table also indicates for each category of safety hole, whether intraprocedural, interprocedural or no analysis is used.

In a few cases, we do not collect safety holes, because the condition seems too common and an error seems relatively unlikely. For example, according to Table 8.1, collecting Free entry safety holes would entail collecting every function that dereferences a pointer argument, as there is no way to check whether a value has been freed. This does not seem useful in practice, and thus SHAna does not collect safety holes in this case. In other cases however, we augment the scope of the categories of safety holes by considering new fault kinds that may arise at the boundary of kernel API functions. For example, the fault types considered by Chou *et al.* [CYC+01] and later by Palix *et al.* [PST+11] only include double-locks and deadlocks in the *Lock* category. Nevertheless, we have seen in mainline bug fixes that attempting to unlock a lock that has not been acquired, or that has been released (double-unlock) is a damaging fault (cf. Section 7.1.3, p. 51). Thus, we also implement search analysis to account for this kind of safety holes. Finally, for the *Null* category of safety holes, SHAna furthermore includes unchecked dereferences of values that in some way depend on the value of an unchecked parameter, and for which the failure caused by the dereference may be hard to diagnose by the service developer. We have already provided an example of such cases in Section 7.1.1 (p. 48).

Intraprocedural In search scenarios that only require intraprocedural analysis, the analyzer scans the definition of the API function to identify code fragments that represent safety holes. For example, in searching for the various *Lock* and *Intr* entry safety holes, SHAna only looks for interrupt disabling operations in the kernel API function itself, because interrupt state flags should not be passed from one function to another [RC01].

Interprocedural In the case of interprocedural analysis, SHAna starts from the definition of an API function and iteratively analyzes all called functions. For example, in searching for *Null* entry safety holes, SHAna searches through both the kernel API function itself and all called functions that receive a parameter of the kernel API function as an argument to find unchecked dereferences.

Category	Actions to avoid faults	safety hole	safety hole description	Analysis type
Block	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held	`entry` `exit`	f calls a blocking function (function referencing GFP_KERNEL) f returns after disabling interrupts or while holding a lock	interprocedural intra/interprocedural
Null	Check potentially NULL/ERR_PTR pointers returned from routines	`entry` `exit`	f dereferences an argument without checking its validity f returns a NULL/ERR_PTR pointer	interprocedural interprocedural
Var	Do not allocate large stack variables ($> 1K$) on the fixed-size kernel stack	`entry` `exit`	f allocates an array whose size depend on a parameter f returns a large value	intraprocedural interprocedural
INull	Do not make inconsistent assumptions about whether a pointer is NULL/ERR_PTR	`entry` `exit`	f dereferences an argument without checking its validity f returns a NULL/ERR_PTR pointer	interprocedural interprocedural
Range	Always check bounds of array indices and loop bounds derived from user data	`entry` `exit`	f uses an unchecked parameter to compute an array index f returns a value obtained from user level	intraprocedural interprocedural
Lock	Released acquired locks; do not double-acquire locks	`entry` `exit`	f acquires a lock derived from a parameter f returns without releasing an acquired lock	interprocedural interprocedural
Intr	Restore disabled interrupts	`entry` `exit`	f calls a blocking function f returns with interrupts disabled	interprocedural intraprocedural
Free	Do not use freed memory	`entry` `exit`	f dereferences a pointer-typed parameter value f frees memory derived from a parameter	none interprocedural
Float	Do not use floating point in the kernel		*These fault kinds depends on local properties and are therefore*	none
Real	Do not leak memory by updating pointers with potentially NULL realloc return values		*not relevant to the interface between a service and the kernel exported functions*	none
Param	Do not dereference user pointers	`entry` `exit`	f dereferences a pointer-typed parameter f returns a pointer-typed value obtained from user level	none interprocedural
Size	Allocate enough memory to hold the type for which you are allocating	`entry` `exit`	f allocates memory of a size depending on a parameter f returns an integer value	intraprocedural none

Table 8.1: Categorization of common faults in Linux [CYC+01, PST+11]. f refers to the *API function*. The *interprocedural* and *intraprocedural* qualifiers indicate whether the analysis is interprocedural or intraprocedural

We now discuss the search strategies that we have implemented for the various categories of safety holes in the taxonomy. Table 8.2 summarizes examples of conditions that enable the safety holes to manifest themselves into faults. SHAna bases its analysis on such *enabling conditions* which provides insights on how to derive the usage preconditions of API functions. The details provided in the following are focused on the identification of safety holes in the implementation of kernel API functions.

Category	Entry safety hole	Enabling condition
Block	f calls a blocking function	service code takes a lock or disables interrupts
Null	f dereferences an argument without checking its validity	service code passes a `NULL` argument to f
Var	f allocates an array whose size depends on a parameter	service code calls f with a large value for that parameter
INull	f dereferences an argument without checking its validity	service code passes a `NULL` argument to f
Range	f uses an unchecked parameter to compute an array index	service code obtains a value from user level
Lock	f acquires lock derived from a parameter	service code re-acquires a lock from an argument to f
Intr	f disables interrupts	service code disables interrupts
Free	f dereferences a pointer-typed parameter value	service code calls or has called a function that may free a value
Param	f dereferences a pointer-typed parameter value	service code forwards a value obtained from user level
Size	f allocates memory of a size depending on a parameter	service code has obtained a value form user level

Table 8.2: Enabling conditions for triggering API function safety holes

Block: To identify API functions that contain *Block* entry safety holes, SHAna performs an interprocedural analysis, searching for functions that may block. In this case, we focus on the common case of functions that are allowed to block during memory allocations. We recognize such functions as those that contain a function call having `GFP_KERNEL` as an argument as this argument allows the basic kernel memory functions to block, waiting for more memory to become available. An extract of analysis specification in SmPL is provided in Appendix B.1 (p. 95).

API functions that contain a *Block* exit safety hole are collected along with the collection of *Lock-/Intr/LockIntr* entry safety holes, which is described below. Along this description will be mentioned a set of relevant functions commonly used for locking (See Appendix A, p. 93).

IsNull/Null: To identify API functions with *IsNull/Null* safety holes, SHAna performs an interprocedural analysis that detects unsafe dereferences of pointer-typed parameters. The search identifies dereferences that are performed prior to any validity check in all, or one, of the control-flow paths. We have also augmented this search strategy by detecting, in the implementation of API functions, unsafe dereferences of other pointer values whose validity depends on the validity of an API function's unchecked parameter. A search specification in SmPL corresponding to that sub-categories of safety holes, and for the most obvious *INull* safety hole, can be found in Appendix B.2 (p. 95).

To collect API functions that exhibit an *IsNull/Null* exit safety hole, SHAna interprocedurally searches for functions that may return an invalid pointer. We recognize both `NULL` and values constructed by the function *ERR_PTR*[1] as invalid pointers. Appendix B.2 (p. 95) provides an example of semantic match for identifying such safety holes in API functions.

Var: To identify API functions that implement *Var* entry safety holes, SHAna performs an intraprocedural analysis for detecting operations of array allocations with sizes depending on a parameter value, whether it is a parameter integer value or an integer derived from a parameter. Appendix B.3 (p. 100) provides an example of search specification matching this safety hole profile.

[1] In this document, we refer to a value constructed with *ERR_PTR* as just "`ERR_PTR`".

To collect API functions with a *Var* exit safety hole, SHAna searches for functions that return large constant values. We parameterize SHAna to report constant integer values larger than 128.

Range: To identify API functions that exhibit *Range* entry safety holes, SHAna performs an intraprocedural analysis for identifying code places where an API function parameter is used to compute an array index.

To collect API functions with a *Range* exit safety hole, SHAna performs an interprocedural analysis for identifying functions that return a value obtained from user-level. The search strategy relies on a number of Linux kernel primitive, listed in Appendix A (p. 93), that give access to user data. We provide an example of semantic match leveraging those primitives in Appendix B.4 (p. 101).

Lock/Intr/LockIntr: To identify API functions that implement a *Lock/Intr/LockIntr* entry safety hole, SHAna performs an intraprocedural analysis, searching for functions that disable interrupts or acquire locks that are derived from parameters. The search strategy relies on a set of commonly used functions for locking and interrupt management listed in Appendix A (p. 93). We have furthermore augmented this category of safety holes by identifying API functions that attempt to release locks that they have not acquired themselves, so as to account for another type of real-world usage bug[2] that were not considered by Chou *et al*. Indeed, while many Linux critical sections both begin and end within the body of a single function, some span function boundaries, implying that some functions expect to be called within a critical section. When the calling code fails to respect this condition, deadlock may ensue.

To collect API functions that expose a *Lock/Intr/LockIntr* exit safety hole, SHAna follows the same search strategy as for *Lock/Intr/LockIntr* entry safety hole, relying on the same locking and interrupt management functions.

Free: To identify API functions that exhibit a *Free* entry safety hole, we consider that any pointer-value argument to an API function may refer to a freed memory, thus making any dereference of a pointer-typed parameter a risky operation. Nonetheless, because practically all API functions dereference their parameters, we do not perform a precondition inference for this sub-category.

To collect API functions that implement the *Free* exit safety hole, SHAna performs an interprocedural analysis for identifying code places where an API function argument is passed to the kernel memory release function `kfree`. Appendix B.6 (p. 105) provides an example of search specification matching this safety hole profile.

Size: To identify API functions that exhibit a *Size* entry safety hole, SHAna relies on an intraprocedural analysis that searches for places where the `kmalloc` and `kzalloc` memory allocation functions are given a size argument involving a `sizeof` expression defined according to an API function parameter value.

8.1.3 Certification Process

As is standard in static analysis, SHAna makes some approximations to ensure termination and scalability. These approximations may result in false positives, *i.e.*, reported safety holes that cannot actually lead to a crash or hang. Using such false positives in the inference of API function usage preconditions may cause two important issues:

1. inferring unnecessary usage preconditions will potentially lead to excessive checking at runtime in the Diagnosys approach, which in turn will degrade the execution performance

[2] Commit: `f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6`

2. false positives may also lead to the generation of runtime warnings that are actually unuseful, and will thus clutter the debug log with messages that are not relevant to any encountered crash or hang.

To address the problem of false positives, our approach requires that a kernel maintainer study the inferred safety holes to discard those that represent false positives. A kernel maintainer is indeed more knowledgeable than a novice service programmer in the internals of core kernel code. Furthermore, the certification process that would consist of validating that the identified safety holes and the inferred preconditions are correct is only necessary for each version of the kernel.

To reduce the workload, SHAna maintains information about safety holes across OS versions, so that the kernel maintainer need only validate reported safety holes in those functions whose definitions have changed. This information is leveraged in a utility that allows for incremental certification between versions of the kernel.

8.2 DIGen: Debugging Interface Generator

DIGen is the counterpart of SHAna for introducing runtime checks and monitoring into the execution of kernel-level services. The main goal of DIGen is thus to allow the creation of new log messages that reflect the usage preconditions of kernel API functions to keep track of any violation of the contracts by service code. To this end, DIGen relies on preconditions uncovered by static analysis to automatically generate and integrate a debugging interface tailored to the implementation of each service under test. This process is illustrated in Figure 8.3.

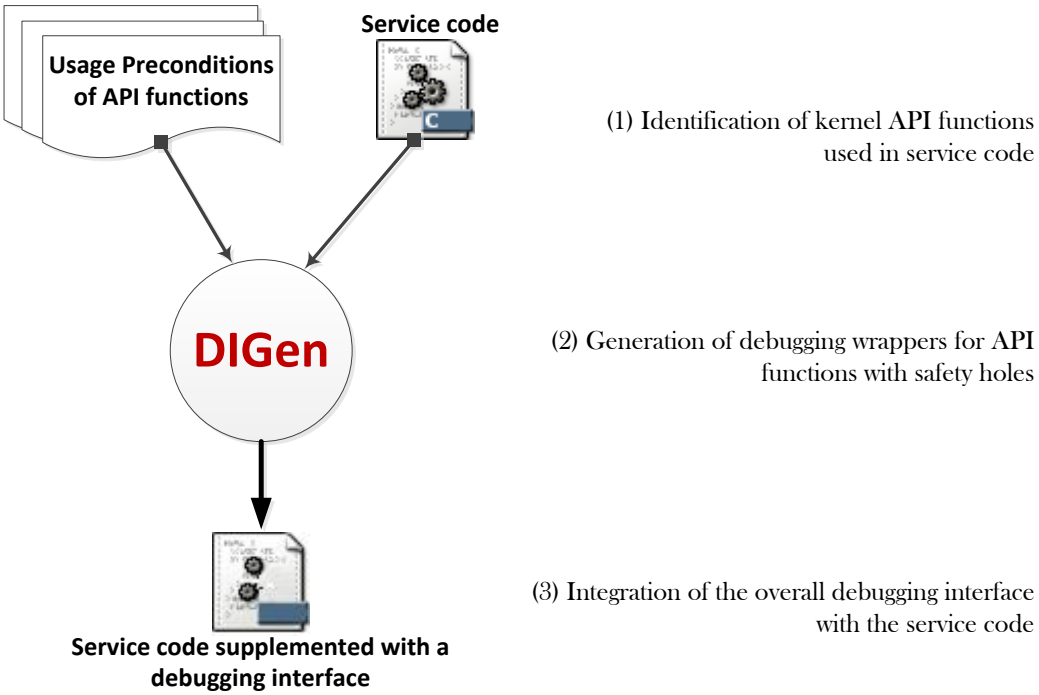

Figure 8.3: The debugging interface generation process

Based on the results of SHAna, DIGen generates a debugging interface as a collection of wrapper functions that augment the definitions of kernel API functions with the necessary checks and calls

to logging primitives, to detect and record violations of safety hole preconditions. Ideally, the kernel maintainer who runs SHAna would also generate a single debugging interface for the entire kernel that could be used by all service developers. Unfortunately, many kernel source files target specific hardware platforms, and thus have mutually incompatible header file dependencies, making it impossible to compile a single debugging interface wrapping all of the kernel API functions at once. Accordingly, we shift the interface generation process into the hands of the service developer, who generates an interface specific to his service. Because the functions invoked by a single service can necessarily be compiled together, this approach avoids all compilation difficulties, while producing a debugging interface that is sufficient for an individual service's needs. We now describe the generation of the debugging interface (Section 8.2.1) and how it is integrated into a service under development (Section 8.2.2).

8.2.1 Generating a debugging interface

The first step in the generation of a debugging interface for a given kernel-level service (e.g., a device driver) is to determine the API functions that are actually called by the service code. A reliable way to identify such functions is to analyze the object code of the tested kernel-level service. DIGen thus disassembles the object code and recovers information about the symbol table entries of the compiled loadable kernel module (*.ko*). Figure 8.4(a) provides an example of a Linux kernel toy module that tests the latency of a *Ftrace* [Ros09] primitive. *Ftrace* is an in-kernel infrastructure for tracing the internal operations of the kernel (e.g., it is meant to accurately answer questions such as "*How long it takes a process to run after it is woken?*" without introducing significant performance penalties in its tracing operations). We also provide in the Figure an excerpt of the symbol table entries obtained from the corresponding module object code (Figure 8.4(b)).

```
#include <linux/init.h>
#include <linux/ftrace.h>
[...]
MODULE_LICENSE("Dual BSD/GPL");
void testing_ftrace () {
    printk (KERN_EMERG "Testing Ftrace\n");
    int i=0;
    for (i=0; i<100000; i++) {
        start = rdtsc();
        trace_printk ("TRACE\n");
        end = rdtsc ();
        latency[i]=end-start;
    }
}
[...]

static int driver_init (void) {
    testing_ftrace (); return 0;
}
static void driver_exit (void) { }
module_init(driver_init);
module_exit(driver_exit);
```

a) Linux toy module for testing Ftrace

```
1  test_ftrace.ko:     file format elf32-i386
2
3  SYMBOL TABLE:
4  [...]
5  00000000 l    d  __mcount_loc   00000000 __mcount_loc
6  00000000 l    d  __versions     00000000 __versions
7  00000000 l    d  .data    00000000 .data
8  [...]
9  00000000 l    df *ABS*  00000000 test_ftrace.c
10 00000000 l     F .text    00000025 driver_exit
11 00000030 l     F .text    00000030 driver_init
12 [...]
13 00000000 l     O .modinfo         00000023 __mod_description10
14 00000023 l     O .modinfo         00000015 __mod_license9
15 00000040 l     O .modinfo         00000034 __mod_author8
16 00000000 l    df *ABS*  00000000 test_ftrace.mod.c
17 [...]
18 00000000 g     F .text    00000025 cleanup_module
19 00000000       *UND*  00000000 __trace_bprintk
20 00000030 g     F .text    00000030 init_module
21 00000000       *UND*  00000000 mcount
22 00000000       *UND*  00000000 printk
```

b) Excerpt of the symbol table entries

Figure 8.4: Identification of core kernel primitives in module object code with GNU *objdump*

The object code represents the symbol table entries from the module object code. In each row[3], the

[3]More information on *GNU objdump* can be found at http://www.gnu.org/software/binutils/.

first number represents the symbol value (i.e., address). The next field is a set of characters and spaces indicating the flag bits that are set on the symbol (e.g., $l \Leftrightarrow local$, $g \Leftrightarrow global$, $u \Leftrightarrow global\ unique$, etc.). Next is the section with which the symbol is associated. When this section is not connected to any other section, i.e., is absolute, the field contains instead *ABS*. However, if the section is referenced in the analyzed object file but not defined there, the field is marked with *UND*. Thus, the *printk*, *mcount* and *__trace_bprintk* symbols which are defined in the core kernel and used in the tested module are marked as undefined (lines 11,13-14.).

Overall, the symbols listed by disassembling the object code not only include symbols that are apparent in the service code (e.g., *printk*), but also include symbols that are called from inside core kernel header files (e.g., *cleanup_module*) or that were integrated after macros expansion (e.g., *__trace_bprintk*). The list of undefined symbols is then mapped with the list of the kernel version exported functions[4]. Based on this method, DIGen can list API functions (which are defined in core kernel) that are called by service code.

For each kernel API function that is used in the service and for which SHAna identified at least one safety hole, DIGen generates a wrapper function. The general structure of such a wrapper is shown in Figure 8.5. Based on the argument values, the wrapper first checks each entry safety-hole precondition (line 4) and then, if the precondition is not satisfied, logs a message indicating the violation. This message includes the safety hole category, which specifies the kind of safety hole and whether the violation is *certain* or *possible* (line 5), as defined in Section 7.2 (p. 52). The wrapper then calls the original function. If the original function has a return value, this value is stored in a local variable, `__ret`, and then the preconditions on any exit safety holes are checked based on this information and on the context (lines 9-10). As all exit safety holes are *possible*, exit safety hole log messages are simply annotated with EXIT (instead of CERTAIN or POSSIBLE for entry safety hole log messages). Finally, the return value, if any, of the original function is returned as the result (line 12).

```
1  static inline ⟨rtype⟩ __debug_⟨kernel function⟩ (...) {
2      ⟨rtype⟩ __ret;
3      /* Check preconditions for entry safety holes */
4      if ⟨an entry safety−hole precondition is violated⟩
5          diagnosys_log(⟨EF id⟩, ⟨SH cat⟩, ⟨info (e.g., arg number)⟩);
6      /* Invocation of the intended kernel function */
7      __ret = ⟨call to kernel function⟩;
8      /* Check preconditions for exit safety holes */
9      if ⟨an exit safety−hole precondition is violated⟩
10         diagnosys_log(⟨EF id⟩, ⟨SH cat⟩, ⟨info (e.g., err ret type)⟩);
11     /* Forward the return value */
12     return __ret;
13 }
14 #define ⟨kernel function⟩ __debug_⟨kernel function⟩
```

Figure 8.5: Wrapper structure for a non-void function

For performance reasons, Diagnosys does not log formatted strings in kernel memory, instead it logs integers representing unique information identifiers that are decoded and translated on-the-fly during log retrieval.

[4]These are the functions that we refer to as API functions throughout the document.

8.2.2 Integrating a debugging interface into a service

Given all API functions with safety holes used in a kernel-level service, DIGen constructs wrappers (one for each API function) and assembles them to implement the generated debugging interface as a header file to be included in the service code. Each wrapper function is followed by a `#define` macro (Figure 8.5, line 14) which, upon expansion, will replace in the service code all calls to kernel API functions with their debugging counterparts. Thus, once compiled with the interface included, the service uses the wrapper functions instead of the corresponding kernel API functions.

To facilitate the integration process of a debugging interface into a kernel-level service under test, Diagnosys provides an automated script, `dmake`. This script manages the generation of the interface in four steps: (1) `dmake` compiles the original service code, (2) identifies the kernel API functions referenced by the resulting object files, (3) generates an interface dedicated to these functions, and (4) recompiles the service with the interface included. The resulting kernel module object thus produced is ready for loading into a running kernel for execution tests.

8.3 CRELSys: Crash-Resilient & Efficient Logging System

To be able to use a Diagnosys-generated debugging interface, the service developer must use a version of the Linux kernel providing support for the Diagnosys runtime system. This support, CRELSys, is a kernel patch, which we have implemented for Linux 2.6.32, that extends the kernel with a crash resilient logging system. The patch additionally configures the kernel to send all crashes and hangs (Linux soft and hard lockups) to the kernel panic function, which the patch extends to reboot into a special *crash kernel* if Diagnosys is activated or to continue with a normal panic, otherwise. Finally, the Diagnosys runtime system includes a tool that can be run from user space to install a copy of the Diagnosys kernel as a crash kernel, initialize the reserved log buffer, activate and deactivate logging, and retrieve and format the logs. We discuss in the following how the logs are stored in the kernel memory (Section 8.3.1) and how they are preserved upon a crash (Section 8.3.2).

8.3.1 Ring-buffer logging in an arbitrary kernel memory area

Once CRELSys has been activated, the service developer may test his code as usual. During service execution, if a wrapper function detects a safety hole for which the precondition is violated, the wrapper logs information about the safety hole in a reserved area of memory, annotated with a timestamp and including the memory address of the call site.

To reserve memory for Diagnosys runtime logs, we leverage the kernel *memmap* boot parameters[5] which allow to instruct the kernel to completely ignore a chunk of memory during its own allocations and use. Thus, CRELSys, which is aware of the location of this memory area will use it at will for the needs of Diagnosys without any risks of conflicts with other parts of the kernel. This reserved area of memory is managed through a ring buffer. When the ring buffer, which is of fixed sized, fills up, adding another log element overwrites the first. In Figure 8.6, the ring buffer is illustrated as able to hold only eight log messages. A ninth message (i.e., *Log 9*) has been added which overwrote the first (i.e., *Log 1*). A subsequent message will overwrite the second (i.e., *Log 2*). Such a ring buffer presents numerous advantages that are very much appreciated in the context of Diagnosys. First, writing to memory is not only fast, compared to doing an I/O to disk, but is also necessary as in Linux programming, only user-level programs should access the file system [KH05]. Second,

[5] For more details see `https://www.kernel.org/doc/Documentation/kernel-parameters.txt`.

continuous logging is practically impossible since memory is limited. Using a ring buffer allows to avoid manually removing logs when memory is filled up. Finally, often, debugging tasks only require to visit the last logs that were inserted into the log buffer. Thus, the ring buffer implemented by CRELSys retains information about only the most recent violations.

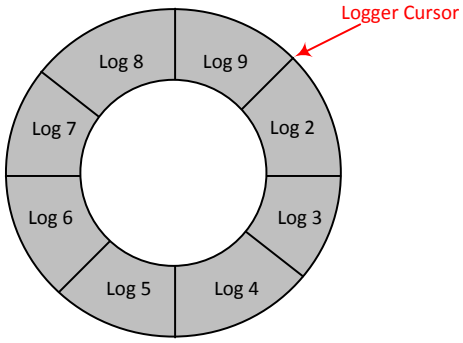

Figure 8.6: A ring buffer for logging

With this architecture based on an early reservation of memory zone for Diagnosys and a careful implementation of a ring-buffer to store the log messages, CRELSys ensures mainly :

- a fast insertion of messages in log buffers to avoid impairing kernel performances
- a reliable retention of critical records on kernel API usage violations

However, upon failure, there is still a need for guaranteeing that the precious information contained in log messages will be available to the developer. This issue is particularly important as operating system failures are often followed by a systematic reboot during which information is lost.

8.3.2 Fast and non-destructive system reboot

On a kernel crash or hang, CRELSys redirects the execution into a *panic* state where using a *Kexec*-based [Nel04] mechanism, the system is rebooted into a new instance of the Diagnosys-enabled kernel. Kexec (i.e., *Kernel execution*) is a mechanism of the Linux kernel that allows some kind of “live” booting of a new kernel over the currently running one.

Figure 8.7 shows the different stages of the Linux system boot process. When a Personal Computer (PC) system is first booted, or is reset, the processor executes code located in the the Basic Input/Output System (BIOS), which is stored on flash memory on the motherboard. The BIOS then must determine which devices are candidates for boot. When a boot device is found, the first-stage bootloader is loaded into RAM and executed. The job of the first bootloader is actually to load the second-stage bootloader into RAM and starts its execution. The second-stage bootloader is then in charge of loading Linux and a temporary root file system (e.g., *ramfs*) into memory. The kernel then takes control to decompress the kernel image and load kernel modules. At the completion of this stage, *init*, the first user-space program is started, and high-level system initialization can be performed.

During system reboot, the bootloader stage is preceded by a shutdown of the previously running system. This involves terminating running processes, writing back cache buffers to disk, unmounting file systems, and performing a hardware reset. The Kexec patch allows to skip the entire bootloader stage and directly jump into the new kernel that the user has programmed to reboot on. Thus the Kexec-based mechanism performs the reboot with no hardware reset, no firmware operation and no

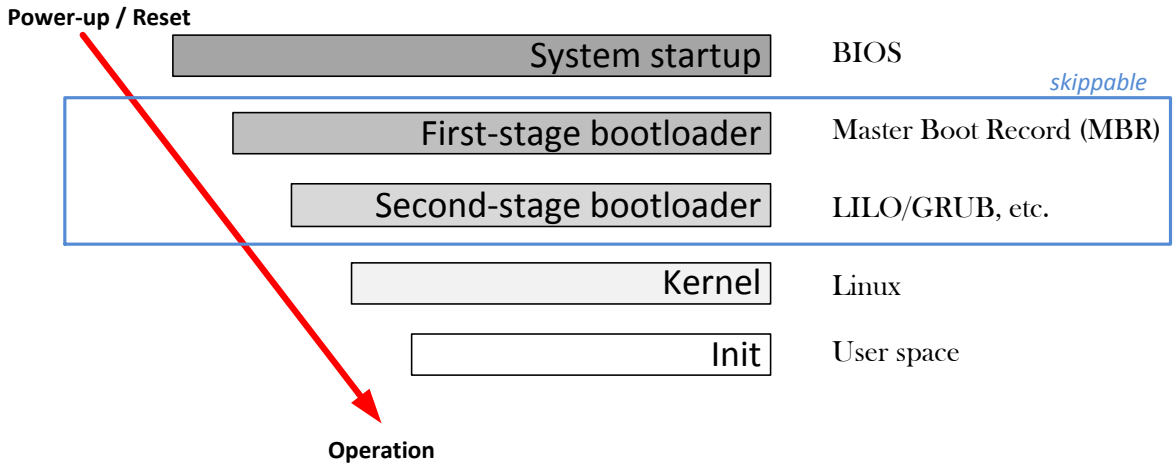

Figure 8.7: Linux boot process

bootloader involved. The main advantage of this process, which is the goal of Kexec, is to increase the speed of the reboot process. We have found however that, because the new kernel we reboot into needs to sit in the same place in memory as the currently running one, the mechanisms is suitable in the scope of CRELSys. Indeed, since the reboot is performed without reinitializing any hardware, including the memory, while the new kernel will still respect the boundaries of the memory reservation, the Kexec mechanism ensures that the accumulated Diagnosys log is still available. The service developer may then access the log messages after a crash that was followed by a fast reboot.

To allow kernel-level service developers to readily access the log messages, upon reboot, from user space, we have implemented a character device driver that transfers data directly to and from a user process. A pseudo device is then setup for this purpose to serve as an entry point of user commands (e.g., *cat /dev/crelsys*) and exit point of log messages that were stored in kernel memory. The messages are made available in the order in which they were generated. When a crash occurs, the Diagnosys runtime system also inserts the kernel stack trace into the Diagnosys log before rebooting.

8.4 Summary

In this chapter, we have detailed the design and some implementation choices of Diagnosys. The Diagnosys approach is implemented through a number of tools for identifying safety holes in kernel code, generating debugging interfaces and supporting a reliable logging.

In section 8.1, we have presented SHAna and discussed the analysis for identifying safety holes in the implementation of API functions, as well as for inferring usage preconditions of these functions. We present in Section 8.2 the process for generating a debugging interface that is tailored to a kernel-level service code. The goal of DIGen is to create new log messages at the boundary between service code and kernel code where they might be the most useful for debugging kernel failures during service code testing. Finally, in Section 8.3 we provide an overview of the CRELSys runtime logging system that will enable developers to always recover the log messages that are produced during service executions and that may contain the cause of the late crash.

Chapter 9

Kernel Debugging with Diagnosys

"It's hard enough to find an error in your code when you're looking for it;
it's even harder when you've assumed your code is error-free."
Steve McConnell

Contents

In this chapter we describe some debugging experiences that we have performed to highlight the benefits of Diagnosys support for developers of kernel-level services. In particular, we wish to investigate how Diagnosys solves the different issues, enumerated and discussed in Section 4.4 (p. 28), that make kernel debugging difficult. Indeed, we have demonstrated that kernel debugging is made difficult by unreliable backtraces and by the questionable relevance of the information in crash reports. Based on the example that were used to explore the need for a new approach to debugging, we assess the qualitative benefits of our proposed Diagnosys approach. To this end, we replay a kernel crash from the `btrfs` file system (Section 9.1). To account for the second category of OS failures, we also replay a hang reported in kernel commit logs (Section 9.2).

9.1 Replaying a kernel crash

Kernel crashes are the most common type of OS failures. When they occur, developers must identify (1) the origin and (2) the cause of the failure based on the oops report that were issued and that they managed to capture, often through heavyweight remote debugging setups. As an example of kernel crash, we again consider the `btrfs` example used for illustration in Section 4.4. Figure 9.1 shows the bug fix patch that were introduced in mainline code to fix the usage of the *open_bdev_exclusive* API function.

For the purpose of this experiment, we have recovered and installed a version of the `btrfs` file system, right before the relevant patch was applied. The obtained code was still compatible to the Linux kernel 2.6.32. The goal of the experiment was then to execute the code so that a fault will manifest itself to reflect the need for the patch. Thus, to cause *open_bdev_exclusive* to fail, we first create and mount a `btrfs` volume and then attempt to add to this volume a new device that is not yet

```
commit 7f59203abeaf18bf3497b308891f95a4489810ad
    bdev = open_bdev_exclusive(...);
-   if (!bdev)
-       return -EIO;
+   if (IS_ERR(bdev))
+       return PTR_ERR(bdev);
```

Figure 9.1: Excerpt of a bug fix patch in `btrfs` file system

created. This, as previously discussed in Section 4.4, leads the *open_bdev_exclusive* API function to return an `ERR_PTR` after failing to locate the device to open.

Figure 9.2 shows the crash report that we have collected from the kernel console at the end of the above experiment. Study of this report in Section 4.4 (p. 28), in the context of debugging, showed that the source of the problem was not readily available in the backtrace. Some of the issues we found were that the backtrace contained many stale pointers that makes it more challenging to readily pinpoint the origin of the crash. Furthermore, the backtrace did not contain information on the root cause of the crash.

```
[  847.353202] BUG: unable to handle kernel paging request at fffffffee
[  847.353205] IP: [<fbc722d9>] btrfs_init_new_device+0xcf/0x5c5 [btrfs]
[  847.353229] *pdpt = 00000000007ee001 *pde = 00000000007ff067 *pte = 0000000000000000
[  847.353233] Oops: 0000 [#1]
[  847.353235] last sysfs file: /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
[  847.353238] Modules linked in: btrfs zlib_deflate crc32c libcrc32c ib_iser rdma_cm ib_cm iw_cm ib_sa [...]
[  847.353271]
[  847.353274] Pid: 3699, comm: btrfs-vol Not tainted (2.6.32-diagnosis-btrfs #32) Latitude E4300
[  847.353276] EIP: 0060:[<fbc722d9>] EFLAGS: 00010246 CPU: 0
[  847.353291] EIP is at btrfs_init_new_device+0xcf/0x5c5 [btrfs]
[  847.353293] EAX: fffffffea EBX: fbc7cc0d ECX: f716ea80 EDX: fbc9cdc0
[  847.353294] ESI: fbc972a0 EDI: 00000004 EBP: f0a61eb8 ESP: f0a61e70
[  847.353296]  DS: 007b ES: 007b FS: 0000 GS: 00e0 SS: 0068
[  847.353298] Process btrfs-vol (pid: 3699, ti=f0a60000 task=ed840ca0 task.ti=f0a60000)
[  847.353299] Stack:
[  847.353301]  fbc98044 ee28e008 ee24bc00 ee31c630 f0a61ebc 00001000 fbc7b84e 00000246
[  847.353304] <0> f0a61ea4 00000000 00000000 f1f62c00 bff0e12c fffffffea c01c52a8 fbc7cc0d
[  847.353308] <0> fbc7cc0d fbc972a0 f0a61ed0 fbc7b87f bff0e12c ee24bc00 ca048334 ee28e000
[  847.353312] Call Trace:
[  847.353327]  [<fbc7b84e>] ? btrfs_ioctl_add_dev+0x33/0x74 [btrfs]
[  847.353334]  [<c01c52a8>] ? memdup_user+0x38/0x70
[  847.353349]  [<fbc7cc0d>] ? btrfs_ioctl+0x0/0x243 [btrfs]
[  847.353363]  [<fbc7cc0d>] ? btrfs_ioctl+0x0/0x243 [btrfs]
[  847.353378]  [<fbc7b87f>] ? btrfs_ioctl_add_dev+0x64/0x74 [btrfs]
[  847.353393]  [<fbc7cdaa>] ? btrfs_ioctl+0x19d/0x243 [btrfs]
[  847.353396]  [<c01f7031>] ? vfs_ioctl+0x21/0x70
[  847.353398]  [<c01f7672>] ? do_vfs_ioctl+0x72/0x580
[  847.353401]  [<c01cbe6e>] ? handle_mm_fault+0x23e/0x9d0
[  847.353404]  [<c01ce635>] ? unmap_region+0xe5/0x100
[  847.353409]  [<c0543a40>] ? do_page_fault+0x160/0x390
[  847.353411]  [<c01f7be7>] ? sys_ioctl+0x67/0x80
[  847.353414]  [<c0108583>] ? sysenter_do_call+0x12/0x28
[  847.353416] Code: 80 b0 1b 00 00 8b 40 6c 85 c0 74 1c c7 45 e0 01 00 00 00 8b 45 e4 83 c0 3c e8 54 [...]
[  847.353433] EIP: [<fbc722d9>] btrfs_init_new_device+0xcf/0x5c5 [btrfs] SS:ESP 0068:f0a61e70
[  847.353449] CR2: 00000000fffffffee
[  847.353451] ---[ end trace 69edaf4b4d3762ce ]---
```

Figure 9.2: Oops report following a `btrfs ERR_PTR` crash in Linux 2.6.32

To compare with information that could potentially be obtained with our approach, we have re-

played the same execution scenario when using Diagnosys. The experiment is identical to the previous one, except that when the kernel crashes, the system is automatically rebooted to make available for exploitation the log messages accumulated by CRELSys. These log messages are actually constructed on the fly, translating identifiers in raw logs to the corresponding meaning based on compiled information from DIGen. A typical Diagnosys log line contains the timestamp of the log, the source file and line number where the unsafe call was performed, the name of the API function, the category of the safety hole and possibly the name of a relevant argument or an unsafe return value. In the case of the replay of the `btrfs` crash, Figure 9.3 shows the last line added to the Diagnosys log before the crash, which is the line that the developer is likely to consult first. This line shows that the function *open_bdev_exclusive* activated an *INull* exit safety hole by returning an `ERR_PTR`. It also reports the runtime timestamp and the call site where the safety hole was violated. Combining this information with the information about the crash site in the oops report and the service source code shows that the problem is the inadequate error handling code after *open_bdev_exclusive*. Using Diagnosys, the service developer can focus on his own code, and does not have to probe the kernel source or object code to obtain the needed information.

```
1 [4294934950]|@/var/diagnosys/tests/my_btrfs/volumes.c:1441|open_bdev_exclusive|INULL(EXITED)|ERR_PTR|
```

Figure 9.3: Last Diagnosys log line in the execution of `btrfs`

9.2 Replaying a kernel hang

Kernel hangs are notoriously hard to debug[1] as they can simply freeze the computer leaving the developer without any information on the ongoing failure. When the kernel is programmed to panic after a certain delay, this panic, which occurs long after the actual fault, can produce a backtrace that is hard to correlate to the source of the problem. In such situations, Diagnosys, which records information about previous potentially dangerous operations, can be a reliable tool for support kernel programmers. We assess the benefits of this support by replaying a bug that were also discussed during the characterization of safety holes in Section 7.1.3 (p. 51).

Just before the release of Linux 2.6.33, the `nouveau_drm` nVidia® graphics card driver contained a hang resulting from the use of the kernel API function *ttm_bo_wait*. This function exhibits a *Lock* entry safety hole and a *Lock* exit safety hole, as it first unlocks and then relocks a lock received via its first argument. The `nouveau_drm` driver called this function without holding this lock, hanging the kernel.

When we do not use the Diagnosys debugging interface, the hang leaves the developer with little information. Using Diagnosys, the hang is immediately detected and causes a kernel panic, which in turn causes a reboot with CRELSys that preserves the execution log messages. In Figure 9.4, the last line of the Diagnosys log shows that *ttm_bo_wait* has been called without the expected lock held. The log messages indicates the type of safety hole, the place of the offending call to the API function and the relevant lock that needs to acquired to avoid the failure.

Correlating the information provided by the Diagnosys log message with the source code suggests taking the lock before the call and releasing it after the call, as shown in the Linux patch in Figure 9.5 which reflects the fix that were ultimately made in mainline code.

[1] See an article at http://www.linuxjournal.com/article/5749

```
1 [437126]|@/var/diagnosys/tests/nouveau/nouveau_gem.c:929|ttm_bo_wait|LOCK/ACQUIRE(POSSIBLE)|bo->lock|
```

Figure 9.4: Last Diagnosys log line in the execution of `nouveau_drm`

```
1 commit f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6
2 if (req->flags & NOUVEAU_GEM_CPU_PREP_NOBLOCK){
3 +   spin_lock(&nvbo->bo.lock);
4       ret = ttm_bo_wait(&nvbo->bo, false, false, no_wait);
5 +   spin_unlock(&nvbo->bo.lock);
6 }
```

Figure 9.5: Patch to avoid a fault involving a Lock safety hole in `nouveau_drm`

9.3 Summary

In this chapter, we have presented two main debugging experiments using the Diagnosys infrastructure. The goal of these experiments was to illustrate the qualitative benefits of the proposed approach. We have mainly showed that, contrary to traditional kernel debugging capabilities with backtraces where the provided information is unreliable and often irrelevant, if not missing, debugging with Diagnosys provides the opportunity to record dangerous operations to help developers more readily pinpoint the mistakes in their code. This approach is even more useful to developers who are not necessarily knowledgeable in the internals of the OS and the idiosyncrasies of kernel API functions.

In the next part of this thesis, we more thoroughly assess the Diagnosys approach both quantitatively. Indeed, we evaluate the extent of the safety hole issue in kernel code and their impact on real-world bugs from mainstream kernel-level services that are used daily in consumer computers. We also evaluate different claims of the approach regarding the benefits in terms of faster and easier debugging, and in terms of performance overheads.

Part III

Assessment

Chapter 10

Opportunity and Benefits of Diagnosys

"You can't trust code that you did not totally create yourself."

Ken Thompson

Contents

In this chapter we assess various aspects of the Diagnosys approach to establish its opportunity in today's context of kernel-level service programming in monolithic commodity operating systems such as Linux. We also perform a quantitative evaluation of the benefits of debugging with Diagnosys. Our experiments use code from Linux 2.6.32, which is used[1] in the 10.04 Long Term Support version of Ubuntu®, in Red Hat Enterprise Linux 6, in Oracle Linux, etc. Our performance experiments are carried out on a Dell 2.40 GHz Intel® Core™ 2 Duo with 3.9 GB of RAM. Unless otherwise indicated, the OS is running a Linux 2.6.32 kernel that has been modified to support CRELSys, the Diagnosys logging infrastructure. 1MB is reserved for CRELSys' crash-resilient log buffer.

The benefit of a Diagnosys-generated debugging interface is determined by the quality of the information collected by SHAna. Thus, Diagnosys is only beneficial if SHAna identifies safety holes in functions that are used by a wide range of drivers and if these functions are likely to be used in an incorrect way. Consequently, we assess in Section 10.1 the number of safety holes collected by SHAna and in Section 10.2 the impact these safety holes have had on the robustness of the Linux kernel. Finally, using fault injection, we assess the completeness of the set of safety holes collected by SHAna.

10.1 Prevalence of safety holes in kernel code

In this section, we investigate how widespread the safety hole types exposed in Section 7.2 (p. 52) are. We describe the exploration of Linux kernel code by SHAna to provide insights on the preva-

[1] At the time of experiments, 2.6.32 was the version used in Ubuntu LTS.

lence of safety holes and the opportunity of devising an approach such as Diagnosys to address the issues that they may pose. Table 10.1 summarizes, for each kind of safety hole, the number of API functions exported in Linux 2.6.32 that SHAna identifies as containing at least one occurrence of that kind of safety hole. In all, SHAna reported 22,940 safety holes in 7,505 exported functions. The most frequently occurring kinds of safety holes are *IsNull/Null*, *Lock/Intr/LockIntr* and *Block*. Over 7,000 functions process pointer-typed parameters without checking their validity. More than 94% of these functions perform unsafe dereferences directly within the body of their definition, and 5% forward the parameter value to other functions that unsafely use them with no prior check. In the *Lock/Intr/LockIntr* entry sub-category, 98% of the over 800 collected functions try to acquire a lock that has been transmitted to them via a parameter, without first checking its state. The remaining 2% assume that the transmitted mutexes or spinlocks are already held in the calling context and unsafely attempt to release them.

Safety hole	Number of exported functions collected in the	
	entry sub-category	exit sub-category
Block	367	815
IsNull/Null	7 220	1 124
Var	5	11
Lock/Intr/LockIntr	815	23
Free	-	11
Size	8	-
Range	-	8

Table 10.1: Prevalence of safety holes in Linux 2.6.32

To estimate the utility of the kernel exported functions in new services, we consider the number of calls to these functions within the kernel code itself. In the 147,403 call sites across the entire kernel source code where exported functions are used, 1 out of 2 calls invokes a function containing a known safety hole. Depending on the kind of safety hole, the median number of calls to functions containing an entry safety hole ranges from 3 to 9, while the median number of calls to functions containing an exit safety hole ranges from 8 to 20. This suggests that the kernel exported functions containing safety holes are likely to be useful to new services.

10.2 Impact of safety holes on code quality

After having established that safety holes are widespread in kernel API functions and that such functions are pervasive in kernel-level service code, we wish to investigate whether API function safety holes are effectively dangerous in kernel programming, i.e., (1) if programmers write programs with bugs that are related to the presence of safety holes in API functions, and (2) if the percentage of those bugs are significant compared to the overall bugs related to the usage of exported functions.

Thus, to assess the impact of the identified safety holes over the course of the development of Linux, we have searched through the changelogs of the history of Linux 2.6[2] to identify patches that mention the kernel API functions exported in Linux 2.6.32. The search strategy consists in considering all commits whose changelogs mention the name of an exported API function. Nonetheless, we ignore commits in which the function name is used as a common word (*e.g.*, “sort”, “panic”, etc.) to limit the number of false positives during manual processing. We have then manually reviewed these patches to

[2] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git - history back to 2.6.12.

identify those that are related to the usage of exported functions. Finally from these relevant patches, we identify those for which the bug fix was made to account for a usage precondition as defined by SHAna. As shown in Table 10.2, 267 out of 703, i.e., 38%, of the usage defects are related to the categories of safety holes that we have classified in this document.

Total number of commits in Linux 2.6	278,078
Commits in any way related to exported functions	11,294
Commits related to the usage of exported functions	703
Commits related to the categorized safety holes	267

Table 10.2: Linux kernel bug fix commits

10.3 Improvement in debuggability

To be useful, Diagnosys must cover a high percentage of the misuses of kernel API functions. We first evaluate this in Section 10.3.1 by artificially creating and activating misuses of API functions in kernel services and measuring how many are trapped by Diagnosys. Additionally, Diagnosys must be able to produce log messages that ease the debugging process. We evaluate the debugging effort in Section 10.3.2 by measuring the number of files and functions that have to be studied to identify the cause of a crash, with and without Diagnosys.

Our experiments involve a number of commonly used kinds of services: networking code, USB drivers, multimedia drivers, and file systems. Services of these kinds make up over a third of the Linux 2.6.32 source code. We have selected a range of services that run on our test hardware. Table 10.3 presents those services along with the number of API functions exhibiting safety holes that they use.

Category	**Service module**	**Description**	**# of used functions with safety holes**
Networking	e1000e	Ethernet adapter	57
	iwlagn	Intel WiFi Next Gen AGN	57
	btusb	Bluetooth generic driver	26
USB drivers	usb-storage	Mass storage device driver	51
	ftdi_sio	USB to serial converter	31
Multimedia device drivers	uvcvideo	Webcam device driver	28
	snd-intel8x0	ALSA driver	35
File systems	isofs	ISO 9660 file system	26
	nfs	Network file system	198
	fuse	File system in userspace	86

Table 10.3: Tested Linux 2.6.32 services

10.3.1 Coverage of Diagnosys

We now assess the number of false negatives of SHAna, *i.e.*, the set of safety holes that can lead to faults in practice but are not identified by SHAna. To determine this coverage of Diagnosys, we first mutate existing services so as to artificially create bugs. Then, we inject faults at run-time to

potentially cause the mutation to trigger actual crashes systematically across the execution of our test services.

Fault model. The largest percentage of our identified safety holes are related to `NULL` and `ERR_PTR` dereferences, and so we focus on these safety holes in our fault injection study. To devise a fault model, we consider how it can happen that such values are manipulated by kernel code. One prominent source of `NULL` and `ERR_PTR` values is to indicate the failure of some sort of allocation. Robust kernel code checks for these values and aborts the ongoing computation. Nevertheless, omission of these tests is common. For example, in Linux 2.6.32, for the standard kernel memory allocation functions *kmalloc*, *kzalloc*, and *kcalloc*, over 8% of the calls that may fail[3] do not test the result before dereferencing the returned value or passing the returned value to another function.

Based on these observations, our fault injection experiments focus on missing `NULL` and `ERR_PTR` tests in the service code. Our mutations remove such tests from the service code, one by one, and use the failslab feature of the Linux fault injection infrastructure [Cor04] within the initialization of the tested value to inject failures into the execution of any call to a basic memory allocation function that this initialization involves. Because the initialization can invoke basic memory allocation functions multiple times, a single mutation experiment may involve multiple injected faults.

Results. One possible result of a fault injection test is that there is no observable effect. This can occur when the code initializing the tested variable does not involve a memory allocation, when the effect of the failure of the memory allocation is confined within the kernel code and does not affect the service, or when the safety hole is *possible* and is not encountered in the actual execution. Another possible result is that there is a crash, but there is no information relevant to the cause of the crash in the Diagnosys log. In this case, either the information has been overwritten in the ring buffer or SHAna has not detected the safety hole, representing a false negative. The final possible result is that there is a crash and information related to the crash is found in the Diagnosys log, representing a success for Diagnosys. In this latter case, we can further consider the position of the information relevant to the crash in the Diagnosys log. It is most helpful for the developer if this information is in the most recent entry before the crash occurred, as this position is easily identifiable.

Table 10.4 presents the fault injection results for 10 services implemented as kernel modules. Overall, we have performed 555 mutations. For each mutation, we have exercised the various execution paths of the affected module. 56% of the experiments have resulted in a service crash. After reboot, in 90% of the cases, the log contained information relevant to the origin of the defect. The table also distinguishes between cases where this information is at the last position in the log buffer and the cases where other information that is irrelevant to the crash was logged subsequently. As a metric of debuggability we use the ratio between the number of crashes for which the log contained information in the last position, and the total number of crashes. On average, Diagnosys has improved the debuggability of the service by 86%. In one case, the improvement is as low as 66%, but there are very few mutation sites in this code.

10.3.2 Ease of the debugging process

Provided with an oops report containing a backtrace and debugging tools that can translate stack entries into file names and line numbers, a developer typically starts from the point of the crash,

[3] Kernel allocation functions use flags to indicate whether the process can afford to have a failed allocation. Calls that are not allowed to fail have the flag information containing `__GFP_NOFAIL` or `__GFP_RETRY`.

Category	Kernel module	# of mutations	# of crashes with			% improved debuggability
			no log	log is not last	log is last	
Networking	e1000e	57	0	0	20	100%
	iwlagn	18	1	0	8	88.9%
	btusb	9	1	0	7	87.5%
USB drivers	usb-storage	11	0	0	3	100%
	ftdi_sio	9	0	0	6	100%
Multimedia device drivers	snd-intel8x0	3	1	0	2	66.7%
	uvcvideo	34	3	3	17	73.9%
File systems	isofs	28	3	0	9	75.0%
	nfs	309	13	9	157	87.7%
	fuse	77	3	1	41	91.1%

Table 10.4: Results of fault injection campaigns

visiting all files and caller functions until the origin of the crash is localized. When the crash occurs deep in the execution, the number of functions and files to visit can become large.

We have considered 199 of the mutations performed in our coverage tests that lead to crashes, from `btusb`, `nfs`, and `isofs`. We also consider 31 mutations in `nfs` code that add statements for arbitrarily acquiring and releasing locks in services in order to provoke kernel hangs, focusing on locks that are passed between functions as they can trigger safety holes in core kernel code.

We have compared the 230 oops reports with the corresponding Diagnosys logs. In 92% of these crashes, the Diagnosys log contains information on the origin of the fault. For those cases, debugging with the oops report alone required consulting 1 to 14 functions, including on average one possibly stale pointer, in up to 4 different files distributed across kernel and service code. In 73% of the cases for which the Diagnosys log contains relevant information, we find that using Diagnosys reduces by at least 50% the number of files and functions to consult. In 19% of the cases for which the Diagnosys log contains relevant information, the crash occurred in the same file as the mutation, but the Diagnosys log made it possible to more readily pinpoint the fault by providing line numbers that are closer to the mutation site.

Finally, we consider the impact of stale pointers on the debugging process. The considered back traces contain an average of 5 entries that are marked as possibly stale, of which on average one appears between the entry indicating the point of crash and the entry of the function where the mutation was performed. We have furthermore assessed the improvement brought by `kdb`, and established that its backtraces contains fewer unreliable entries, but still include 2 on average.

Our assessment has also shown that kernel backtraces can miss functions, which can be attributed, in some cases, to tail call optimizations. Such corrupted stack traces can then adversely affect debugging.

10.4 Summary

In this chapter, we have investigated a posteriori the need for the Diagnosys approach to deal with the prevalence of safety holes in kernel code (Section 10.1, p. 77). Based on bug fixes submitted to the Linux mainline kernel, we establish that safety holes, especially those that we have defined in our taxonomy, have a real impact on the quality of kernel-level services (Section 10.2, p 78).

In section 10.3 (p. 79), we have quantified the improvement in debuggability brought by Diagnosys. Relying on a fault model that concerns a common type of faults we have established that the analysis of SHAna leaves very few false negatives, i.e., most safety-hole related faults will be noticed by Diagnosys during execution. We have also estimated the gain in debugging by showing that our approach reduces the effort for tracing back an error and pinpointing the cause of a fault.

Chapter 11

Overheads

Contents

The benefits of Diagnosys come at the cost of some overhead, both in terms of the effort required for the kernel maintainer to certify the safety holes identified by SHAna, and in terms of the runtime cost of the checks and logging operations performed by the debugging interface during service execution. We show that the information maintained by SHAna substantially reduces the effort required from the kernel maintainer over time, while the service execution overhead is minimal. Section 11.1 assesses the overhead incurred by the certification of the analysis results provided by SHAna. Section 11.2 details the runtime overheads introduced by the debugging interfaces. The measurements include both micro and macro benchmarks.

11.1 Certification overhead of analysis results

Static analysis is necessarily approximate, as it does not have complete access to run-time values. This may lead to false positives, in which a safety hole is reported that in fact cannot lead to a crash. Such false positives can increase the logging time and clutter the log with irrelevant messages.

The Linux operating system provides more than 10,000 configuration features which can be combined at compile-time. Those configurations, whose combinations can be problematic [TLSSP11], are often incompatible, making a few execution paths unlikely, in general or for a given execution context. To account for false positive safety holes that are due to the presence of multiple, configuration-specific, definitions of some functions SHAna annotates as potential false positives, safety holes derived from calls to such functions with the file in which the relevant function instance is defined.

Of the 22,940 safety holes reported by SHAna for Linux 2.6.32, SHAna itself annotated 465 (2%) as potential false positives, because of the ambiguity of the identification of called functions during interprocedural analysis. Since the Linux kernel provides different definitions of some functions for

different architectures, these different definitions may exhibit different safety holes, and therefore require a thorough validation of analysis results. We have attempted to evaluate the necessary delay for checking the annotated results of SHAna. At a rate of about 5 minutes per safety hole, this certification requires about a week of work (38 hours). Of the 465 potential safety holes, we have found that 405 (87%) are actual false positives.

We have also manually reviewed all other reported safety holes by SHAna for Linux 2.6.32. Among the remaining reported safety holes that were not annotated as potential false positives during the analysis, we have identified some cases for which misuse seems very unlikely. For example, some lock-related exported functions such as *unlock_rename* clearly indicate their purpose in their name. Similarly, *clk_get_rate* may return a large integer, but it seems unlikely that a developer would use this integer to declare the size of an array. We have found 9 such false positives in Linux 2.6.32. Most of the associated functions are called fewer than 5 times, with the most frequently used, *clk_get_rate*, being called 144 times. Thus, given the small rate of these safety holes and the low usage of the associated functions, we consider that it is sufficient for the kernel maintainer to certify the safety holes annotated as potential false positives by SHAna.

To further reduce the certification overhead, Diagnosys maintains information about safety holes across OS versions, so that the kernel maintainer need only validate reported safety holes in those functions whose definitions have changed. To demonstrate the potential benefit of this information, we have also certified the SHAna annotated safety holes in 5 versions that were released after Linux 2.6.32. As shown in Figure 11.1, the burden on the maintainer is significantly reduced when data from a previous certification are available. Between two certification processes, the workload can drop by 50 to 95%, often to around a day or less, depending on the amount of time elapsed since the release of the previously certified version.

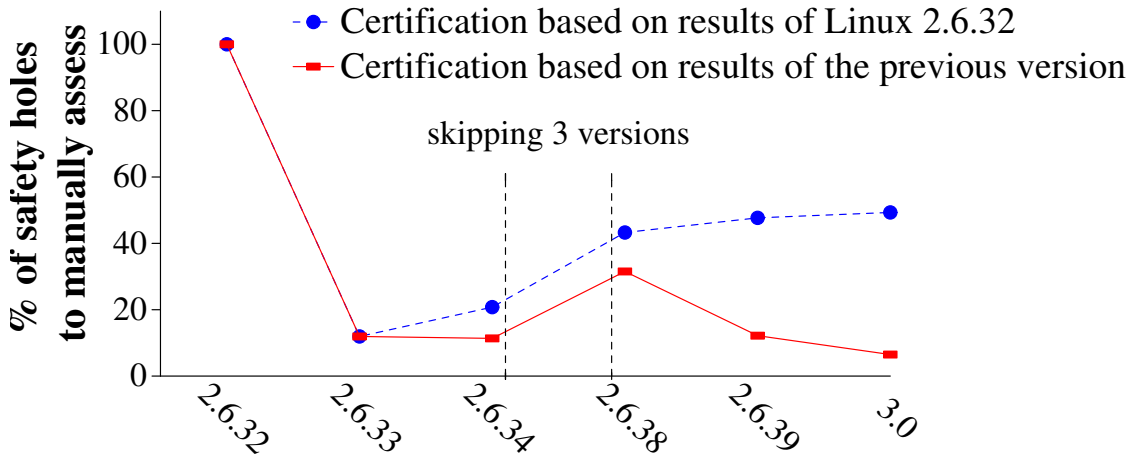

Figure 11.1: Certification overhead

Considering that the releases of Linux distributions are not aligned with the versionning of the Linux kernel, service developers may have to go from programming drivers for one kernel version (e.g., Linux 2.6.32) to programming for another (e.g., Linux 2.6.38). Thus the analysis may be skipped for the intermediary versions (e.g., Linux 2.6.33 to 2.6.37). We note that even in such cases, performing a certification based on a previous, but not preceding, version can still substantially reduce the maintainer workload.

11.2 Service execution overhead

Testing preconditions and logging incur a performance overhead on the execution of a kernel-level service. This overhead must be sufficiently small to avoid interfering with the normal service execution. In this section, we evaluate the overheads introduced by the primitives used by Diagnosys to test preconditions (Section 11.2.1), and investigate at a macroscopic level the impact of Diagnosys on service performance (Section 11.2.2).

11.2.1 Penalties introduced by Diagnosys primitives

To measure the execution time of the Diagnosys precondition checking and logging operations, we have used the *Klogger* framework [ETKF][1], a state-of-the-art tool for performing fine grained logging of kernel operations. We also compare the execution time of a call to an exported API function with an empty body to that of a call to an exported API function containing a single precondition test. Table 11.1 summarizes the overhead for one instance of a each of the types of validity tests performed by a Diagnosys debugging interface. The observed overhead varies between 1.35% and 11.04%.

Check	**Primitive**	**Performance** (processor clock ticks)	**Overhead (%)**
Pointer validity	*IS_ERR_OR_NULL*	248.13 ± 121.24	3.12%
Spin_lock state	*spin_is_locked*	267.19 ± 121.24	11.04%
Mutex state	*mutex_is_locked*	243.88 ± 109.13	1.35%
Interrupt state	*irqs_disabled*	260.66 ± 91.34	8.32%
Performance of a call to an exported function with an empty body			240.62 ± 95.19

Table 11.1: Checking overhead $\pm$ standard deviation

Table 11.2 compares the execution time of Diagnosys' logging primitive with that of other logging mechanisms used in the kernel. *printk* is the most commonly used logging function. *Ftrace* [Ros09] optimizes the logging process by deferring formatting from tracing time to output time. In Diagnosys, string formatting is not needed as the log message is generated at compile-time and only contains integers that uniquely identify either an API function, a safety hole type, etc. Diagnosys' logging primitive is 1.3x faster than Ftrace's *trace_printk*, and 5x faster than *printk*. In Diagnosys, the time-consuming processing tasks are performed in user space once the service developer attempts to display the log messages.

Logger	printk	Ftrace (trace_printk)	Diagnosys
Execution time (processor clock ticks)	3280.05 ± 82.52	884.16 ± 578.124	673.15 ± 129.26

Table 11.2: Performance of the Diagnosys logging primitive

[1] Klogger kernel patch for Linux 2.6.31.4

11.2.2 Impact of Diagnosys on service performance

To understand the global performance overhead induced by the Diagnosys approach, we execute various real-world kernel services with and without a generated debugging interface.

Network driver performance. Our first test scenario involves a Gigabit Ethernet device that requires both low latency and high throughput to guarantee high performance. We evaluate the impact of a debugging interface by exercising the *e1000e* Linux device driver using the `TCP_STREAM`, `UDP_STREAM` and `UDP_RR` tests from the *netperf* benchmark [Jon]. For these experiments, the *netperf* utility was configured to report results accurate to 5% with 99% confidence. Table 11.3 summarizes the performance and CPU overhead for the *e1000e* driver when it is run without and with a debugging interface. The debugging interface only reduces the throughput by 0.4% to 6.4%, and increases the CPU utilization by 0.4% to 10%. Nevertheless, while small, the existence of this overhead suggests why kernel developers would not want to systematically implement API functions such that they always perform all of these checks. This shows the need for a pluggable debugging interface dedicated to a service under development, as provided by Diagnosys.

Test		Without Diagnosys	With Diagnosys	Overhead
`TCP_STREAM`	Throughput	907.91 Mb/s	904.32 Mb/s	0.39%
	CPU	52.57%	58.48%	10.10%
`UDP_STREAM`	Throughput	951.00 Mb/s	947.73 Mb/s	0.34%
	CPU	58.92%	65.45%	9.98%
`UDP_RR`	Throughput	7371.69 Tx/s	6902.81 Tx/s	6.36%
	CPU	55.19%	55.37%	0.33%

Table 11.3: Performance of the `e1000e` driver

File system performance. Our second test scenario involves the NFS file system, whose implementation uses about 200 exported functions exhibiting safety holes. The experiment consists of sequential read, sequential write/rewrite and random seek phases based on patterns generated by the Bonnie benchmark [Bra]. For this experiment, the client and server run on the same machine, connected using a network loopback interface, to eliminate the network transmission time. During a run of this benchmark with a debugging interface integrated into the `NFS` file system, we have recorded over 48,000,000 calls to the interface wrapper functions to write and read 8G of data. As shown in Table 11.4, for data transfers of only one character, amounting to 1 byte, the overhead can be significant, of up to 67%. For block reads and writes, however, the overhead is only up to 17%, and for random seeks and sequential rewrites it is under 3%.

11.3 Summary

In this chapter we have investigated the overheads incurred by the Diagnosys approach at different levels. First, we evaluate in Section 11.1 the maintainer burden. We show that thanks to information stored by SHAna, the certification process is significantly eased, especially when certified results from a previous version of the kernel are available. In Section 11.2 (p. 85), we assess the impact of Diagnosys' debugging interfaces on the execution of kernel-level services. After micro-benchmarks

Test		**Without Diagnosys** (Access rate - K/sec)	**With Diagnosys** (Access rate - K/sec)	**Overhead**
Sequential reads	per char	930	642	30.9%
	per block	28795	23811	17.3%
Sequential writes	per char	494	162	67.2%
	per block	42467	38329	9.7%
Sequential rewrites		13647	13327	2.3%
Random seeks		2145	2143	0.9%

Table 11.4: Performance of the `NFS` file system

have established that the implementation choices in Diagnosys are favorable to performance, we use a driver for a Gigabit Ethernet device and a NFS file system to show that the performance impact of our approach is within the limits of what is acceptable when testing a kernel-level service in the initial stages of development. The experiments also show that the approach can even be used up to the phase of initial deployment.

Chapter 12

Conclusion

Defects in kernel-level services can cause the demise of the entire operating system, often leaving developers without any clue as to what went wrong. In the Linux kernel for example, one significant difficulty in developing drivers is that the kernel does not export a debugging interface to its internal functionalities [KH]. Many of the functions that are exported to external modules have implicit ill-documented preconditions, which, if not satisfied, can cause the entire system to crash or hang.

In this thesis, we have presented a new approach for supporting kernel-level service developers in early-stages of development. Diagnosys [BRLM12] was designed and implemented as an approach to automatically constructing a debugging interface for the Linux kernel. The approach is based on static analysis of the kernel code, to infer the preconditions of each API function, making it possible to construct an interface containing a wrapper for each identified function in use by service code. This constructed debugging interface performs a runtime monitoring of the interactions between the services and the kernel. The Diagnosys approach is therefore set in several phases:

- First, a designated kernel maintainer uses Diagnosys to identify constraints in the use of kernel API functions.
- Second, based on this information, developers of kernel-level services can then use Diagnosys to generate a debugging interface specialized to their code.
- Third, when a service including this interface is tested, it records information about potential problems. This information is preserved following a kernel crash or hang.

Our experiments show that the generated debugging interface provides useful log information and incurs a low performance overhead.

In the remainder of this chapter, we present a summary of the different contributions of our thesis before outlining some ongoing and future work in line with the Diagnosys approach.

12.1 Contributions

In the course of building this thesis, we have made different contributions on better understanding some causes of the difficulties of writing safe and reliable kernel-level services (Section 12.1.1). We have also proposed a practical approach that even novice service programmer, without substantial knowledge on kernel code, could use to produce better driver code (Section 12.1.2). We have finally evaluated our work, by 1) demonstrating the acuteness of the safety hole issue and discussing the

benefits of the Diagnosys approach (Section 12.1.3), and by 2) validating its practical usefulness and usability (Section 12.1.4).

12.1.1 Characterization of safety holes

In this thesis, we have discussed the notion of *safety hole*, a fragment of code that introduces a potential of fault in the interaction between a driver and the kernel. We have then provided a characterization of safety holes in the kernel programming interface. We have furthermore highlighted how safety holes pose a serious problem to kernel developers. Finally, we have provided an extensive exploration of the static analysis rules that we have written to identify instances of safety holes that reflect the common faults that appear in the Linux kernel.

12.1.2 Diagnosys

The Diagnosys approach is the central element of our thesis. We have proposed an automated tool for identifying safety holes and monitoring the execution of relevant code to improve testing tasks in the early stages of kernel-level service development. The debugging interface constructed by Diagnosys amounts to a collection of wrappers on the kernel API functions, that logs information about the potentially dangerous uses of these functions. Localizing the interface in this way, at the boundary of the interface between the service and the OS kernel, ensures that the feedback provided by the interface is in terms of the code that the developer has written, and that he is thus expected to be familiar with. In this thesis, we describe in details the implementation of Diagnosys, including the static analysis phase, the runtime checks, and the logging operations. We further describe the usage steps of Diagnosys for testing and debugging a given kernel-level service.

12.1.3 Debugging benefits

We have conducted an assessment of the Diagnosys approach and of its implementation using both qualitative and quantitative measurements. The aim of this assessment was to show the benefits of Diagnosys in terms of debugging improvement.

Qualitatively, we have demonstrated that the log of dangerous operations recorded by Diagnosys contains reliable information about the crash of the kernel. For this evaluation, we have performed experiments that consisted on replaying real faults that were reported to kernel developers.

Quantitatively, we have evaluated the improvement of Diagnosys on the debugging of kernel-level services through mutation testing. We have mutated 10 kernel-level services from various categories, representative of the major part of services written by kernel developers. The services, which include device drivers and file systems, were mutated to get safety hole-related faults to be manifested during execution. These experiments allowed to assess the capability of Diagnosys to help developers quickly pinpoint the locations of these faults in service code.

12.1.4 Limited overheads

Diagnosys comes with a number of overheads, both with respect to the certification of the static analysis results, the runtime checks and the logging operations introduced by the debugging interface. We have measured these different overheads to show that the Diagnosys approach is viable. Furthermore, we have evaluated the overall impact of Diagnosys on service performance based on the stressed execution of two services, namely a Gigabit Ethernet device driver and a network file system. The experiments revealed that the performance degradation with Diagnosys is minimal.

12.2 Ongoing and future work

The work presented in this document describes a new, practical, hybrid approach — based on static code analysis and runtime monitoring — for supporting kernel-level service developers in their testing and debugging tasks. With Diagnosys [BRLM12], we have provided different tools at different steps of the approach, and some of those tools can be adapted to other software engineering issues.

Ongoing work. Building on the Diagnosys approach, we are currently attempting to generalize to other, including non-OS, software. Indeed, as discussed in Section 2.3 (p. 10), many software are built based on the plug-in model, and as such, present the parallel issues. Plug-ins interacting with core software can lead to different execution errors at the boundary of the APIs that need to be monitored. To this end, our first endeavor is to automatically infer the interface of a software that is available for use by external code and identify the relevant safety holes that appear in the interface. We have already collected hundreds of open source software projects for this new study.

Future work. Debugging approaches, such as Diagnosys, are most relevant when software bugs appear before the code is shipped. Unfortunately, the rapid-pace of modern software produces a huge amount of bugs which are reported back to developers for fixing. Development teams manage to fix and document an important number of bugs. More importantly, as Andy Chou, co-designer of the Coverity static analysis tool [ECCH00], has stated about the deluge of buggy mobile software, the exposed bugs are nothing new and are "actually well-known and well-understood in the development community – the same use after free and buffer overflow detects we have seen for decades". We have witnessed this fact in the evaluation of the impact of safety holes on the quality of kernel-level services: the different faults are actually similar and based on the same implicit preconditions or the same programming errors. Nevertheless, when project teams, such as the Linux kernel maintainers team, are flooded with bug reports, the time-to-fix interval increases and some even remain unfixed for a very long time. It is thus necessary to start investigating a new approach to drastically improve the fix rate of bugs through automatic bug fix recommendations. To this end, we could rely on a collected knowledge on safety hole categories and leverage historical information extracted from software development artifacts, including bug reports and bug fix links, to produce candidate fixes for newly reported bugs. Complementary to Diagnosys, this approach would rely on information retrieval techniques that have already been shown successful for helping bug triagers detect and dismiss duplicate bug reports. We thus take these techniques further to accelerate, and if possible avoid, manual debugging by presenting developers with fix directions.

12.3 Concluding remarks

Diagnosys was designed as a complementary approach to various research work [LBP+09, SBL03, MRC+00]. Indeed, in practice, current approaches have not been completely effective in clearly solving the need for producing more reliable kernel-level services. On one hand, the standard practice remains to write device drivers by hand despite the existence of techniques for generating them [MRC+00, RCKH09]. On the other hand, device driver developers, who are not experts in kernel programming, cannot afford to thoroughly validate their code before shipping their services, which leads to the presence of about 600 to 700 faults in each release of the kernel [PST+11].

With the Diagnosys approach, we propose to support developers by providing them with runtime debugging support, and post-mortem code analysis information that will allow even novice program-

mers to pinpoint the origin and understand the cause of some important, and idiosyncratic, programming faults [BRLM12]. This approach has received academic praise[1] from researchers in both software engineering and systems reliability communities. We have also been contacted by developers of device drivers at Intel® who have asked assistance in using Diagnosys in their development process. This manifestation of interest thus shows that the Diagnosys approach deals with an important problem in kernel-level service development. We hope to continue investigating around the problem and improving the proposed solution.

[1] Best paper award at the 27^{th} IEEE/ACM International conference on Software Engineering (ASE 2012)

Appendix A

Relevant kernel primitives

The Linux kernel provides various commonly known primitives for different programming tasks. We provide in this Appendix a list of those that we have collected and used in the implementations of SHAna.

A.1 Locking functions

{mutex,spin,read,write}_lock
{mutex,spin,read,write}_trylock

A.2 Interrupt management functions

cli, local_irq_disable

A.3 Functions combining locking and interrupt management

{read,write,spin}_lock_irq
{read,write,spin}_lock_irqsave, local_irq_save, save_and_cli

A.4 Kernel/userland access primitives

getuser, memcpy_fromfs, copy_from_user

Appendix B

Examples of semantic matches implemented in SHAna

The semantic matches illustrated in this appendix are provided for describing search specifications implemented in SHAna. They include different SmPL rules for matching and reporting identified safety holes from the various categories of our taxonomy.

Contents

B.1 *Block* safety holes

We provide in Figure B.1 an excerpt of the semantic match implemented in SHAna for detecting *Block entry* safety holes in Linux kernel API functions. In this specification, SHAna focuses on API functions that call kernel primitives with the `GFP_KERNEL` flag enabled, allowing the kernel to block. The analysis in this case is interprocedural to collect all API functions that may block after calling internal routines that can block.

B.2 *Null/INull* safety holes

For *Null/INull* safety holes, we present in Figure B.2 (p. 97) the complete version of the semantic match that we have used to describe the capabilities of Coccinelle in Section 5 (p. 31). As discussed previously, this semantic match collects kernel API functions that directly dereferences their arguments without any validity check.

Figure B.3 (p. 98) details the semantic match for a more subtle kind of *Null/INull entry* safety hole where the unchecked value that is dereferenced is derived from an unchecked parameter. We

```
1  virtual start
2  virtual after_start
3  virtual should_be_static
4  #include "block.ml"
5
6  @export depends on !start && !after_start@
7  identifier handler;
8  declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
9  @@
10 (
11  EXPORT_SYMBOL(handler);
12 |
13  EXPORT_SYMBOL_GPL(handler);
14 )
15 @exported@
16 identifier export.handler;
17 position p;
18 @@
19 handler@p (...) {...}
20
21 @script:ocaml@
22 symbol << export.handler; p << exported.p; sf << virtual.save_file;
23 version << virtual.save_file; linux << virtual.linux;
24 @@
25 start_iteration symbol (List.hd p).file version linux sf
26
27 // ---------- COMING back to start ----------
28 @r0 depends on !should_be_static@
29 identifier virtual.fn;
30 @@
31 static fn (...) {...}
32
33 @r depends on (start || (after_start && !should_be_static && !r0) || (after_start && should_be_static)) exists@
34 identifier virtual.fn; position p1, p2;
35 @@
36 fn@p1 (...) {
37 <+... GFP_KERNEL@p2 ...+>
38 }
39
40 @forwards depends on !r && (start || (after_start && !should_be_static && !r0) || (after_start && should_be_static)) exists@
41 identifier virtual.fn, in_fn; position p, p2;
42 @@
43 fn@p (...) {
44 ... when any
45 in_fn@p2 (...)
46 ... when any
47 }
48
49 //We eliminate some trivial functions
50 @is_special depends on forwards@
51 identifier special ~= ".*lock\|.*unlock\|.*irq_save\|.*irq_restore\|wait_event"; position forwards.p2;
52 @@
53 special@p2 (...)
54
55 @script:ocaml depends on is_special@
56 @@
57 Coccilib.include_match(false)
```

Figure B.1: Detecting *Block entry* safety hole instances – This semantic match is focused on *possible* safety holes

have discussed in Section 8.1.2 (p. 60) how errors related to this kind of safety holes are hard to debug. Contrary to the case of the previous semantic match, we have implemented this one to perform an interprocedural analysis.

Finally, we present in Figure B.4 (p. 99) an excerpt of the semantic match implemented for identifying *Null/INull exit* safety holes interprocedurally. The search consists in detecting API functions that can explicitly return an invalid pointer : `NULL` or an `ERR_PTR` value.

```
1  #include "direct_deref.ml"
2
3  @export@
4  identifier handler;
5  declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
6  @@
7  (
8      EXPORT_SYMBOL(handler);
9  |
10     EXPORT_SYMBOL_GPL(handler);
11 )
12
13 @exist_deref depends on export exists@
14 type T;
15 position p;
16 expression new_val;
17 identifier param, fld;
18 identifier export.handler;
19 parameter list [h_number] params_prec;
20 @@
21 handler(params_prec, T *param, ...){
22 ... when != param = new_val
23     when != param == NULL
24     when != param != NULL
25     when != IS_ERR(param)
26     param->fld@p
27 ... when any
28 }
29
30 @forall_deref@
31 type exist_deref.T;
32 position exist_deref.p;
33 expression new_val;
34 identifier exist_deref.param, exist_deref.fld;
35 identifier export.handler;
36 parameter list [h_number] exist_deref.params_prec;
37 @@
38 handler(params_prec, T *param, ...){
39 ... when != param = new_val
40     when != param == NULL
41     when != param != NULL
42     when != IS_ERR(param)
43     when forall
44     param->fld@p
45 ... when any
46 }
```

Figure B.2: Detecting obvious intraprocedural *Null/INull entry* safety hole instances

```
virtual check_internal
virtual check_dependancy
#include "subtle_deref.ml"
@exported depends on !check_internal && !check_dependancy@
identifier handler;
declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
@@
(
    EXPORT_SYMBOL(handler);
|
    EXPORT_SYMBOL_GPL(handler);
)
@export_def@
identifier exported.handler; position p;
@@
handler@p (...) {...}
@fn_uses_arg depends on !check_dependancy exists@
type T;  expression arg, E, val, new_val, other_val;
identifier param, fn; identifier virtual.handler;
parameter list [h_number] params_prec;
position p1, p2;     expression list [f_number] args_prec;
@@
handler(params_prec, T* param, ...){
... when != param = new_val
    when any
(
E = fn@p1(args_prec, <+... param ...+>, ...);
|
arg=<+... param ...+>;
... when != arg = val
    when any
E = fn@p1(args_prec, <+... arg ...+>, ...);
)
... when != E == NULL
    when != E != NULL
    when != IS_ERR(E)
    when != E = other_val
(
    E == NULL
|
    E != NULL
|
    IS_ERR(E)
|
    E = other_val
|
    E@p2
)
... when any
}
// When invalid pointer E2 is returned, it constitutes another
// safety hole reported in another category
@returned depends on fn_uses_arg exists@
position fn_uses_arg.p2; identifier virtual.handler;
expression fn_uses_arg.E;
@@
handler(...){
...
return E@p2;
}
@only_pointers depends on fn_uses_arg && !returned@
type ret_T; ret_T *pointer_global; position fn_uses_arg.p1;
identifier pointer_local, fn_uses_arg.fn;
@@
(
    pointer_global = fn@p1(...);
|
    ret_T *pointer_local = fn@p1(...);
)
@static_fn depends on fn_uses_arg@
identifier fn_uses_arg.fn;
@@
static fn(...){...}
// ----- Coming back for check_dependancy -----
@found_fn depends on check_dependancy@
identifier virtual.fn; position p_def;
@@
fn@p_def (...){...}
@fn_fails depends on check_dependancy exists@
identifier virtual.fn, fld; expression E, ret; position p, found_fn.p_def;
@@
fn@p_def(...){
    ... when any
(
    return@p NULL;
|
    return@p ERR_PTR(...);
|
    ret@p = NULL;
    ... when != (ret=E | ret->fld)
    return ret;
|
    ret@p = ERR_PTR(...);
    ... when != (ret=E | ret->fld)
    return ret;
)
...
}
@fails_following_param depends on fn_fails exists@
type T; position p, found_fn.p_def;
expression new_val; identifier virtual.fn, param;
parameter list [number] params_prec;
@@
fn@p_def (params_prec, T param, ...){
... when != param = new_val
    (param@p || ...)
    ...
    return ...;
}
@other_fails_following_param depends on fn_fails exists@
type T; position p, found_fn.p_def; expression E, new_val;
identifier virtual.fn, param; parameter list [number] params_prec;
@@
fn@p_def (params_prec, T param, ...){
... when != param = new_val
    E = <+... param ...+>;
    ...
    (E@p || ...)
    ...
    return ...;
}
```

Figure B.3: Detecting subtle *Null/INull entry* safety hole instances of type *possible*

```
1  // May return NULL or ERR_PTR
2  virtual after_start
3  virtual should_be_static
4  #include "return_null.ml"
5
6  @ret_ptr@
7  type T;
8  identifier virtual.fn;
9  @@
10 T *fn (...) {...}
11
12 @export depends on ret_ptr && !after_start@
13 identifier virtual.fn; declarer name EXPORT_SYMBOL; declarer name EXPORT_SYMBOL_GPL;
14 @@
15 (
16     EXPORT_SYMBOL(fn);
17 |
18     EXPORT_SYMBOL_GPL(fn);
19 )
20
21 @r0 depends on ret_ptr && !should_be_static@
22 identifier virtual.fn;
23 @@
24 static fn (...) {...}
25
26 // fn may be the exported symbol or a function whose
27 // return value is returned by the exported function
28 @ret_inv_ptr depends on ret_ptr && (export || (after_start && !should_be_static && !r0) || (after_start && should_be_static)) exists@
29 position p, p1; identifier virtual.fn; expression *E; expression E0;
30 @@
31 fn@p (...) {
32 ... when any
33 (
34     return@p1 NULL;
35 |
36     return@p1 ERR_PTR (...);
37 |
38     E = NULL
39     ... when != E = E0
40         return@p1 E;
41 |
42     E = ERR_PTR(...)
43     ... when != E = E0
44         return@p1 E;
45 )
46 }
47
48 @forward depends on !ret_inv_ptr && ret_ptr && (export || (after_start && !should_be_static && !r0) || (after_start && should_be_static)) exists@
49 position p; expression E; expression *ret; identifier virtual.fn, in_fn;
50 @@
51 fn (...) {
52 ... when any
53 (
54     return@p in_fn (...);
55 |
56     ret@p = in_fn (...)
57     ... when != ret = E
58         return ret;
59 )
60 }
```

Figure B.4: Detecting *Null/INull exit* safety hole instances

B.3 *Var* safety holes

Figure B.5 shows the semantic match that tracks *Var entry* safety hole instances in kernel API functions. The specification is written to detect array allocations that are performed based on the value of an API function parameter.

```
1  #include "var.ml"
2
3  @export@
4  identifier fn;
5  declarer name EXPORT_SYMBOL;
6  declarer name EXPORT_SYMBOL_GPL;
7  @@
8  (
9  EXPORT_SYMBOL(fn);
10 |
11 EXPORT_SYMBOL_GPL(fn);
12 )
13
14 @r@
15 identifier export.fn;
16 identifier i, param;
17 parameter list [n] paramsb;
18 type T, T1, T2;
19 T1 V;
20 expression E;
21 position p, p1;
22 @@
23 fn(paramsb, T param@p1, ...) {
24 ... when any
25     when forall
26 (
27     T1 i[sizeof(...)];
28 |
29     T1 i@p[<+... param ...+>];
30 |
31     V = <+... param ...+>;
32     ... when != V = E
33         T2 i@p[<+... V ...+>];
34 )
35 ... when any
36 }
37
38 @script:ocaml@
39 n << r.n;
40 p2 << r.p;
41 param << r.param;
42 symbol << export.fn;
43 sf << virtual.save_file;
44 vers << virtual.version;
45 @@
46 let nb = n+1 in
47 report_var_safety symbol param nb (List.hd p2).file (List.hd p2).line vers sf "certain"
48
```

Figure B.5: Detecting *Var entry* safety hole instances of type *certain*

B.4 *Range* safety holes

We present in Figure B.6 the specification implemented in SHAna for identifying kernel API functions that contain a *Range exit* safety hole. The search, which is performed intraprocedurally, consists of detecting cases where an API function may return a value obtained from userland.

```
#include "range.ml"

@export@
identifier fn;
declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
@@
(
    EXPORT_SYMBOL(fn);
|
    EXPORT_SYMBOL_GPL(fn);
)

@c@
identifier export.fn;
expression E, E0, V, V0;
position p, p2;
type T;
@@
fn (...) {
... when any
(
copy_from_user@p((T)E,...)
|
copy_from_user@p(E,...)
|
memcpy_fromfs@p(E,...)
|
memcpy_fromfs@p((T)E,...)
)
... when != E = E0
(
    return@p2 <+... E ...+>;
|
    V = <+... E ...+>;
    ... when != V = V0
    return@p2 <+... V ...+>;
)
}

@get_prim@
identifier primitive;
position c.p;
@@
primitive@p(...)

@script:ocaml@
symbol << export.fn;
p << c.p;
sf << virtual.save_file;
primitive << get_prim.primitive;
vers << virtual.version;
@@
let p_=List.hd p in
report_range_safety symbol primitive p_.file p_.line vers sf "may"
```

Figure B.6: Detecting *Range exit* safety hole instances of type *possible*

B.5 *Lock/Intr/LockIntr* safety holes

In the category of *Lock/Intr/LockIntr* safety holes, different situations may appear as the primitives vary. We discuss in this Appendix the *exit* safety holes that consist for an API function to take a lock or systematically disables interrupts while finishing its execution.

Figure B.7 shows the semantic match for detecting API functions that may take a lock (and omit to release it) until it returns. The primitives used in this case are for locking only. Figure B.8 (p. 103) on the other hand presents the specification for collecting API functions that systematically leave the interrupts disabled after they have been called. In Figure B.9 (p. 104)we provide the semantic match that searches for cases where the API function both takes a lock and disables interrupts before returning.

```
1  #include "lock.ml"
2
3  @exported@
4  identifier handler;
5  declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
6  @@
7  (
8      EXPORT_SYMBOL(handler);
9  |
10     EXPORT_SYMBOL_GPL(handler);
11 )
12
13 @locked depends on exported@
14 position p,p1;
15 expression E1;
16 @@
17 (
18 mutex_lock@p1
19 |
20 mutex_trylock@p1
21 |
22 spin_lock@p1
23 |
24 spin_trylock@p1
25 |
26 read_lock@p1
27 |
28 read_trylock@p1
29 |
30 write_lock@p1
31 |
32 write_trylock@p1
33 ) (E1@p,...);
34
35 @ends_in_lock exists@
36 expression locked.E1;
37 identifier lock;
38 position locked.p,p1,p2;
39 identifier exported.handler;
40 @@
41 handler (...) { <+...
42 lock@p1 (E1@p,...);
43 ... when != E1
44 return@p2 ...;
45 ...+> }
46 @ends_in_unlock depends on ends_in_lock exists@
47 expression locked.E1;
48 identifier unlock;
49 position p!=locked.p;
50 identifier exported.handler;
51 @@
52
53 handler (...) {
54 <+...
55 unlock (E1@p,...);
56 ... when != E1
57 return ...;
58 ...+>
59 }
60
61 @balanced depends on ends_in_lock && ends_in_unlock exists@
62 position locked.p, p1 != locked.p1;
63 identifier ends_in_lock.lock,ends_in_unlock.unlock, exported.handler;
64 expression E,locked.E1;
65 @@
66 handler (...) { <+...
67 if (E) {
68  <+... when != E1
69  lock(E1@p,...)
70  ...+>
71 }
72 ... when != E1
73      when forall
74 if (E) {
75  <+... when != E1
76  unlock@p1(E1,...)
77  ...+>
78 }
79 ...+> }
80
81 @script:ocaml depends on ends_in_lock && ends_in_unlock
82                                        && !balanced@
83 p << locked.p;
84 symbol << exported.handler;
85 sf << virtual.save_file;
86 version << virtual.version;
87 lock << ends_in_lock.lock;
88 @@
89 let p_ = List.hd p in
90 report_lock_safety symbol lock p_.file p_.line "may" version sf
```

Figure B.7: Detecting *Lock exit* safety hole instances of type *possible*

```
#include "intr.ml"

// This virtual rule is never used
virtual after_start

@exported@
identifier handler;
declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
@@
(
    EXPORT_SYMBOL(handler);
|
    EXPORT_SYMBOL_GPL(handler);
)
// For recent 2.6 versions, save_and_cli was deprecated
// and now removed from kernel
@locked depends on exported@
position p,p1;
expression E1;
@@
(
local_irq_save@p1
|
save_and_cli@p1
) (E1@p,...);

@ends_disabling exists@
expression locked.E1;
identifier lock;
position locked.p,p1,p2;
identifier exported.handler;
@@

handler (...) { <+...
lock@p1 (E1@p,...);
... when != E1
return@p2 ...;
...+> }

@ends_reenabling depends on ends_disabling exists@
expression locked.E1; identifier unlock;
position p!=locked.p; identifier exported.handler;
@@
handler (...) { <+...
unlock (E1@p,...);
... when != E1
return ...;
...+> }

@balanced depends on ends_disabling && ends_reenabling exists@
position locked.p, p1 != locked.p1; expression E,locked.E1;
identifier ends_disabling.lock,ends_reenabling.unlock,
    exported.handler;
@@
handler (...) { <+...
if (E) {
 <+... when != E1
 lock(E1@p,...)
 ...+>
}
... when != E1
    when forall
if (E) {
 <+... when != E1
 unlock@p1(E1,...)
 ...+>
}
...+> }

@script:ocaml depends on (ends_disabling && !ends_reenabling)
                        || (ends_disabling && !balanced)@
lock << ends_disabling.lock;
sf << virtual.save_file; version << virtual.version;
p << locked.p; symbol << exported.handler;
@@
let p_ = List.hd p in
report_intr_safety symbol lock p_.file p_.line "arg" "may" version sf
```

Figure B.8: Detecting *Intr exit* safety hole instances of type *possible*

```
#include "lock_intr.ml"

@exported@
identifier handler;
declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
@@
(
    EXPORT_SYMBOL(handler);
|
    EXPORT_SYMBOL_GPL(handler);
)

@locked depends on exported@
position p,p1;
expression E1;
@@
(
spin_lock_irq@p1
|
read_lock_irq@p1
|
write_lock_irq@p1
|
spin_lock_irqsave@p1
|
read_lock_irqsave@p1
|
write_lock_irqsave@p1
)
(E1@p,...);

@ends_locked_and_disabling exists@
position locked.p,p1,p2;
identifier exported.handler;
expression locked.E1; identifier lock;
@@
handler (...) { <+...
lock@p1 (E1@p,...);
... when != E1
return@p2 ...;
...+> }

@ends_unlocked_and_reenabling depends on
          ends_locked_and_disabling exists@
expression locked.E1; identifier unlock;
position p!=locked.p; identifier exported.handler;
@@
handler (...) { <+...
unlock (E1@p,...);
... when != E1
return ...;
...+> }

@balanced depends on ends_locked_and_disabling &&
                     ends_unlocked_and_reenabling exists@
expression E,locked.E1; position locked.p, p1 != locked.p1;
identifier ends_locked_and_disabling.lock,
    ends_unlocked_and_reenabling.unlock, exported.handler;
@@
handler (...) { <+...
if (E) {
 <+... when != E1
 lock(E1@p,...)
 ...+>
}
... when != E1
    when forall
if (E) {
 <+... when != E1
 unlock@p1(E1,...)
 ...+>
}
...+> }

@script:ocaml depends on (ends_locked_and_disabling      &&
!ends_unlocked_and_reenabling) || (ends_locked_and_disabling
                                                  && !balanced)@
p << locked.p; symbol << exported.handler;
sf << virtual.save_file; version << virtual.version;
lock << ends_locked_and_disabling.lock;
@@
let p_ = List.hd p in
report_lock_intr_safety symbol lock p_.file p_.line "may" version sf
```

Figure B.9: Detecting *LockIntr exit* safety hole instances of type *possible*

B.6 *Free* safety holes

Figure B.10 describes the semantic match implementing in SHAna the search for *Free exit* safety holes. This search, which is performed interprocedurally, is programmed to detect cases where an API might return a pointer to a memory that has been freed with `kfree`.

```
virtual start
virtual after_start
virtual should_be_static

#include "free.ml"

// Just match exported functions and iterate
@export depends on !start && !after_start@
identifier fn;
declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
@@
(
    EXPORT_SYMBOL(fn);
|
    EXPORT_SYMBOL_GPL(fn);
)

@def depends on export@
identifier export.fn; position p;
@@
fn@p (...){...}

@script:ocaml@
symb << export.fn; sf << virtual.save_file;
dir << virtual.linux; p << def.p;
version << virtual.version;
@@
start_iteration symb dir (List.hd p).file version sf

// ------- START | AFTER_START
//   Make sure you do not confuse static functions
@r0 depends on !should_be_static@
identifier virtual.fn;
@@
static fn (...) {...}

@fns depends on (start || (after_start &&
 !should_be_static && !r0) || (after_start &&
  should_be_static)) exists@
expression E, E0, E1; identifier virtual.fn;
parameter list [n] paramsb; type T;
identifier i; position p, p2;
@@

fn@p(paramsb, T i, ...) {
... when != \(i = E\|&i\)
(
    kfree@p2 (<+... i ...+>);
|
    E0 = <+... i ...+>;
    ... when != E0 = E1
    kfree@p2 (E0);
)
 ... when any
}

//If the params number do not match why bother
@script:ocaml depends on after_start@
n1 << fns.n; n2 << virtual.fn_number;
@@
let n1 = n1 +1 in let n2 = int_of_string n2 in
if (n1 <> n2) then
Coccilib.include_match(false)

@forw depends on !fns && (start ||
  (after_start && !should_be_static && !r0) ||
    (after_start && should_be_static)) exists@
identifier virtual.fn, i, in_fn;
type T; position p, p2; expression E, E0, E1;
parameter list [n] paramsb; expression list [ne] argsb;
@@
fn@p(paramsb, T i, ...) {
... when != \(i = E\|&i\)
(
    in_fn@p2 (argsb, <+... i ...+>, ...);
|
    E0 = <+... i ...+>;
    ... when != E0 = E1
    in_fn@p2 (argsb, E0, ...);
)
 ... when any
}

@script:ocaml depends on after_start@
n1 << forw.n; n2 << virtual.fn_number;
@@
let n1 = n1 +1 in let n2 = int_of_string n2 in
if (n1 <> n2) then
Coccilib.include_match(false)

@in_fn_static@
identifier forw.in_fn;
@@
in_fn(...) {...}

@script:ocaml depends on after_start && forw@
symb << virtual.symb; fn << virtual.fn;
c_file << virtual.call_file;
e_file << virtual.export_file; p << forw.p;
@@
add_for_filtering symb fn e_file c_file (List.hd p).file

@script:ocaml depends on forw && start && !in_fn_static@
e_file << virtual.export_file;
fn << forw.in_fn; symb << virtual.symb;
dir << virtual.linux; save << virtual.save_file;
n1 << forw.ne; n2 << forw.n;
version << virtual.version; param << forw.i;
@@
let n1 = n1 + 1 in let n2 = n2 + 1 in
external_search symb fn param n2 n1 dir e_file symb version save
```

Figure B.10: Detecting *Free exit* safety hole instances of type *possible*

B.7 *Size* safety holes

To detect *Size entry* safety holes, we rely on the semantic match of Figure B.11. The specification searches for locations in kernel API functions where the size of memory to be allocated with common kernel allocation primitives is computed following the value of a kernel API function parameter.

```
1  #include "size.ml"
2
3  @exported@
4  identifier fn;
5  declarer name EXPORT_SYMBOL, EXPORT_SYMBOL_GPL;
6  @@
7  (
8      EXPORT_SYMBOL(fn);
9  |
10     EXPORT_SYMBOL_GPL(fn);
11 )
12
13 @r exists@
14 type T, T1, T2;
15 T2 **z;
16 expression x, y, E;
17 identifier param, exported.fn;
18 parameter list [number] params_b;
19 position p;
20 typedef u8;
21 {void *, char *, unsigned char *, u8 *} a;
22 {struct device, struct net_device} dev;
23 @@
24 fn (params_b, T param, ...) {
25 ...
26 (
27     z = \(kmalloc\|kzalloc\)(<+... sizeof(T1) ...+>, ...);
28 |
29     a = \(kmalloc\|kzalloc\)(<+... sizeof(T1) ...+>, ...);
30 |
31     x@p = \(kmalloc\|kzalloc\)(<+...sizeof(*param) ...+>, ...);
32 |
33     y = <+... param ...+>;
34     ... when != y = E
35     x@p = \(kmalloc\|kzalloc\)(<+... sizeof(*y) ...+>, ...);
36 )
37 ... when any
38 }
39
40 @script:ocaml@
41 symb << exported.fn;
42 p << r.p;
43 sf << virtual.save_file;
44 vers << virtual.version;
45 param << r.param;
46 n << r.number;
47 @@
48 let number = n+1 in
49 let p_=List.hd p in
50 report_size_safety symb param number p_.file p_.line vers sf "may"
```

Figure B.11: Detecting *Size entry* safety hole instances of type *possible*

Bibliography

[AAF04] Arnaud ALBINET, Jean ARLAT, et Jean-Charles FABRE. « Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel ». In *DSN'04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 867–876, Florence, Italy, 2004.

[AHM+08] Nathaniel AYEWAH, David HOVEMEYER, J. David MORGENTHALER, John PENIX, et William PUGH. « Using Static Analysis to Find Bugs ». *IEEE Software*, 25:22–29, 2008.

[AM03] Karine ARNOUT et Bertrand MEYER. « Uncovering Hidden Contracts: The .NET Example ». *Computer*, 36:48–55, 2003.

[BA08] Suhabe BUGRARA et Alex AIKEN. « Verifying the Safety of User Pointer Dereferences ». In *IEEE Symposium on Security and Privacy*, pages 325–338, Oakland, CA, USA, 2008.

[BBC+06] Thomas BALL, Ella BOUNIMOVA, Byron COOK, Vladimir LEVIN, Jakob LICHTENBERG, Con MCGARVEY, Bohus ONDRUSEK, Sriram K. RAJAMANI, et Abdullah USTUNER. « Thorough Static Analysis of Device Drivers ». In *EuroSys'06: Proceedings of the 2006 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 73–85, Leuven, Belgium, 2006.

[BDH+09] Julien BRUNEL, Damien DOLIGEZ, René Rydhof HANSEN, Julia L. LAWALL, et Gilles MULLER. « A foundation for flow-based program matching: using temporal logic and model checking ». In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 114–126, Savannah, GA, USA, 2009.

[Ber92] Brian N. BERSHAD. « The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems ». In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–211, 1992.

[Bis10] Tegawendé F. BISSYANDÉ. « Language-based Approach for Software Specialization ». In *EuroSys Doctoral Symposium*, pages 1–2, Paris, France, février 2010.

[BMMR01] Thomas BALL, Rupak MAJUMDAR, Todd MILLSTEIN, et Sriram K. RAJAMANI. « Automatic predicate abstraction of C programs ». In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, Snowbird, Utah, United States, 2001.

[BPS00] William R. BUSH, Jonathan D. PINCUS, et David J. SIELAFF. « A static analyzer for finding dynamic programming errors ». *Softw. Pract. Exper.*, 30(7):775–802, juin 2000.

[Bra] Tim BRAY. « The Bonnie File System Benchmark ». http://www.textuality.com/bonnie/.

[BRLM12] Tegawendé F. BISSYANDÉ, Laurent RÉVEILLÈRE, Julia L. LAWALL, et Gilles MULLER. « Diagnosys: automatic generation of a debugging interface to the Linux kernel ». In *ASE'12: Proceedings of 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 60–69, Essen, Germany, 2012.

[BT05] S. BAILEY et T. TALPEY. « The Architecture of Direct Data Placement (DDP) and Remote Direct Memory Access (RDMA) on Internet Protocols ». RFC 4296, 2005.

[CB93] J. Bradley CHEN et Brian N. BERSHAD. « The impact of operating system structure on memory system performance ». In *SOSP'93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 120–133, Asheville, North Carolina, United States, 1993.

[CC10] Vitaly CHIPOUNOV et George CANDEA. « Reverse engineering of binary device drivers with RevNIC ». In *EuroSys'10: Proceedings of the 2010 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 167–180, Paris, France, 2010.

[CCM+09] Miguel CASTRO, Manuel COSTA, Jean-Philippe MARTIN, Marcus PEINADO, Periklis AKRITIDIS, Austin DONNELLY, Paul BARHAM, et Richard BLACK. « Fast byte-granularity software fault isolation ». In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, Big Sky, Montana, USA, 2009.

[CNR09] Domenico COTRONEO, Roberto NATELLA, et Stefano RUSSO. « Assessment and Improvement of Hang Detection in the Linux Operating System ». In *SRDS'09: Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, pages 288–294, Niagara Falls, NY, USA, 2009.

[Cor04] Jonathan CORBET. « Injecting faults into the kernel ». http://lwn.net/Articles/209257/, November 2004.

[CW02] Hao CHEN et David WAGNER. « MOPS: an infrastructure for examining security properties of software ». In *CCS'02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, Washington, DC, USA, 2002.

[CYC+01] Andy CHOU, Junfeng YANG, Benjamin CHELF, Seth HALLEM, et Dawson ENGLER. « An Empirical Study of Operating Systems Errors ». In *SOSP'01: Proceedings of the 18th ACM Symposium on Operating System Principles*, pages 73–88, Banff, Canada, 2001.

[dGdS99] Juan-Mariano de GOYENECHE et Elena Apolinario Fernandez de SOUSA. « Loadable Kernel Modules ». *IEEE Software*, 16:65–71, 1999.

[DLS02] Manuvir DAS, Sorin LERNER, et Mark SEIGLE. « ESP: path-sensitive program verification in polynomial time ». In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, Berlin, Germany, 2002.

[ECCH00] Dawson ENGLER, Benjamin CHELF, Andy CHOU, et Seth HALLEM. « Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions ». In *OSDI'00: Proceedings of the 2000 Symposium on Operating System Design & Implementation*, pages 1–16, San Diego, CA, USA, 2000.

[ECH+01] Dawson ENGLER, David Yu CHEN, Seth HALLEM, Andy CHOU, et Benjamin CHELF. « Bugs as deviant behavior: a general approach to inferring errors in systems code ». In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, Banff, Alberta, Canada, 2001.

[EPG+07] Michael D. ERNST, Jeff H. PERKINS, Philip J. GUO, Stephen MCCAMANT, Carlos PACHECO, Matthew S. TSCHANTZ, et Chen XIAO. « The Daikon system for dynamic detection of likely invariants ». *Sci. Comput. Program.*, 69:35–45, December 2007.

[ETKF] Yoav ETSION, Dan TSAFRIR, Scott KIRKPATRICK, et Dror G. FEITELSON. « Fine grained kernel logging with KLogger: experience and insights ». In *EuroSys'07: Proceedings of the 2007 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 259–272, Lisbon, Portugal.

[FL] Cormac FLANAGAN et K. Rustan M. LEINO. « Houdini, an Annotation Assistant for ESC/Java ». In *FME'01*, pages 500–517, London, UK.

[FLL+02] Cormac FLANAGAN, K. Rustan M. LEINO, Mark LILLIBRIDGE, Greg NELSON, James B. SAXE, et Raymie STATA. « Extended static checking for Java ». In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, Berlin, Germany, 2002.

[FX03] C. FETZER et Zhen XIAO. « HEALERS: a toolkit for enhancing the robustness and security of existing applications ». In *DSN'03: Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pages 317–322, San Francisco, CA, USA, 2003.

[Gra90] J. GRAY. « A census of Tandem system availability between 1985 and 1990 ». *Reliability, IEEE Transactions on*, 39(4):409 –418, oct 1990.

[HHL+97] Hermann HÄRTIG, Michael HOHMUTH, Jochen LIEDTKE, Jean WOLTER, et Sebastian SCHÖNBERG. « The performance of μ-kernel-based systems ». In *SOSP'97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, Saint Malo, France, 1997.

[Hoa69] C. A. R. HOARE. « An axiomatic basis for computer programming ». *Commun. ACM*, 12(10):576–580, octobre 1969.

[HPSA10] Robert HIRSCHFELD, Michael PERSCHEID, Christian SCHUBERT, et Malte APPELTAUER. « Dynamic contract layers ». In *SAC'10: Proceedings of the 2010 Symposium on Applied Computing*, pages 2169–2175, Sierre, Switzerland, 2010.

[HR00] M. HUTH et M. RYAN. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[IBM01] IBM. « Linux Watch », October 2001.

[Jon] Rick JONES. « Netperf: A Network Performance Benchmark, version 2.4.5 ». http://www.netperf.org.

[KCC10] Volodymyr KUZNETSOV, Vitaly CHIPOUNOV, et George CANDEA. « Testing Closed-Source Binary Device Drivers with DDT ». In *ATC'10: USENIX Annual Technical Conference*, Boston, MA, USA, 2010.

[Ker10] Michael KERRISK. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st édition, 2010.

[KH] Greg KROAH-HARTMAN. « The Linux Kernel Driver Interface (all your questions answered and then some) ». http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt.

[KH05] Greg KROAH-HARTMAN. « Driving Me Nuts - Things You Should Never Do in the Kernel ». *Linux Journal*, (133):9, May 2005.

[LBP+09] Julia L. LAWALL, Julien BRUNEL, Nicolas PALIX, René Rydhof HANSEN, Henrik STUART, et Gilles MULLER. « WYSIWIB: A Declarative Approach to Finding API Protocols and Bugs in Linux code ». In *DSN'09: Proceedings of the 2009 International Conference on Dependable Systems and Networks*, pages 43–52, Lisbon, Portugal, 2009.

[LBP+12] Julia L. LAWALL, Julien BRUNEL, Nicolas PALIX, René Rydhof HANSEN, Henrik STUART, et Gilles MULLER. « WYSIWIB: exploiting fine-grained program structure in a scriptable API-usage protocol-finding process ». *Software: Practice and Experience*, 2012.

[LLH+10] J. LAWALL, B. LAURIE, R.R. HANSEN, N. PALIX, et G. MULLER. « Finding Error Handling Bugs in OpenSSL Using Coccinelle ». In *EDCC'10: Proceedings of the European Conference on Dependable Computing*, pages 191 –196, Valencia, Spain, april 2010.

[LLMZ04] Zhenmin LI, Shan LU, Suvda MYAGMAR, et Yuanyuan ZHOU. « CP-Miner: a tool for finding copy-paste and related bugs in operating system code ». In *OSDI'04: Proceedings of the 2004 Symposium on Operating System Design & Implementation*, pages 289–302, San Francisco, CA, 2004.

[LUSG04] Joshua LEVASSEUR, Volkmar UHLIG, Jan STOESS, et Stefan GÖTZ. « Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines ». In *OSDI'04: Proceedings of the 2004 Symposium on Operating System Design & Implementation*, pages 17–30, San Francisco, CA, USA, 2004.

[LZ05] Zhenmin LI et Yuanyuan ZHOU. « PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code ». In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, Lisbon, Portugal, 2005.

[MC11] Paul MARINESCU et George CANDEA. « Efficient Testing of Recovery Code Using Fault Injection ». *ACM Transactions on Computer Systems (TOCS)*, 29(3), novembre 2011.

[Mer01] Stephan MERZ. « Model Checking: A Tutorial Overview ». In *MOVEP'00: Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, pages 3–38, Nantes, France, 2001.

[Mey88] Bertrand MEYER. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st édition, 1988.

[Mil04] Charlie MILLS. *Using Design by Contract in C*. OnLamp.com, O'Reilly, 1st édition, October 2004.

[MRC+00] F. MÉRILLON, L. RÉVEILLÈRE, C. CONSEL, R. MARLET, et G. MULLER. « Devil: An IDL for Hardware Programming ». In *OSDI'00: Proceedings of the 2000 Symposium on Operating System Design & Implementation*, pages 17–30, San Diego, California, 2000.

[MSZ09] Aravind MENON, Simon SCHUBERT, et Willy ZWAENEPOEL. « TwinDrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest OS drivers ». In *ASPLOS'09: Proceedings of the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, Washington, DC, USA, 2009.

[Nel04] Hariprasad NELLITHEERTHA. « Reboot Linux faster using kexec ». http://www.ibm.com/developerworks/linux/library/l-kexec/index.html, 2004.

[PK07] Hendrik POST et Wolfgang KÜCHLIN. « Integrated static analysis for Linux device driver verification ». In *IFM'07: Proceedings of the 6th international conference on Integrated formal methods*, pages 518–537, Oxford, UK, 2007.

[PLHM08] Yoann PADIOLEAU, Julia L. LAWALL, René Rydhof HANSEN, et Gilles MULLER. « Documenting and automating collateral evolutions in Linux device drivers ». In *EuroSys'08: Proceedings of the 2008 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 247–260, Glasgow, Scotland, 2008.

[PLM06] Yoann PADIOLEAU, Julia L. LAWALL, et Gilles MULLER. « Understanding collateral evolution in Linux device drivers ». In *EuroSys'06: Proceedings of the 2006 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–71, Leuven, Belgium, 2006.

[PLM10] Nicolas PALIX, Julia LAWALL, et Gilles MULLER. « Tracking code patterns over multiple software versions with Herodotos ». In *AOSD'10: Proceedings of the 2010 International Conference on Aspect-Oriented Software Development*, pages 169–180, Rennes and Saint-Malo, France, 2010.

[PST+11] Nicolas PALIX, Suman SAHA, Gaël THOMAS, Christophe CALVÈS, Julia Laetitia LAWALL, et Gilles MULLER. « Faults in Linux: Ten Years Later ». In *ASPLOS'11: Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, USA, 2011.

[RC01] Alessandro RUBINI et Jonathan CORBET. « *Linux Device Drivers* », Page 109. O'Reilly Media, second édition, 2001.

[RCK+09] Leonid RYZHYK, Peter CHUBB, Ihor KUZ, Etienne LE SUEUR, et Gernot HEISER. « Automatic device driver synthesis with Termite ». In *SOSP'09: Proceedings of the 2009 ACM symposium on Operating systems principles*, pages 73–86, Big Sky, MN, USA, 2009.

[RCKH09] Leonid RYZHYK, Peter CHUBB, Ihor KUZ, et Gernot HEISER. « Dingo: Taming Device Drivers ». In *EuroSys'09: Proceedings of the 2009 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 275–288, Nuremberg, Germany, 2009.

[RGJ07] Murali Krishna RAMANATHAN, Ananth GRAMA, et Suresh JAGANNATHAN. « Path-Sensitive Inference of Function Precedence Protocols ». In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 240–250, Minneapolis, MN, USA, 2007.

[Ros09] Steven ROSTEDT. « Debugging the kernel using Ftrace ». http://lwn.net/Articles/365835/, December 2009.

[SBL03] Michael M. SWIFT, Brian N. BERSHAD, et Henry M. LEVY. « Improving the Reliability of Commodity Operating Systems ». In *SOSP'03: Proceedings of the 2003 ACM symposium on Operating systems principles*, pages 207–222, Bolton Landing, NY, USA, 2003.

[Stu08] Henrik STUART. « Hunting Bugs with Coccinelle ». Master's thesis, Department of Computer Science, University of Copenhagen, Denmark, 2008.

[TLSSP11] Reinhard TARTLER, Daniel LOHMANN, Julio SINCERO, et Wolfgang SCHRÖDER-PREIKSCHAT. « Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem ». In *EuroSys'11: Proceedings of the 2011 ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–60, Salzburg, Austria, 2011.

[TZM+08] Lin TAN, Xiaolan ZHANG, Xiao MA, Weiwei XIONG, et Yuanyuan ZHOU. « AutoISES: automatically inferring security specifications and detecting violations ». In *SS'08: Proceedings of the 17th USENIX conference on Security symposium*, pages 379–394, San Jose, CA, 2008.

[YHD+] Mao YANDONG, Chen HAOGANG, Zhou DONG, Wang XI, Zeldovich NICKOLAI, et Kaashoek M. FRANS. « Software Fault Isolation with API integrity and multi-principal modules ». In *SOSP'11: Proceedings of the 2011 ACM symposium on Operating systems principles*.

[YMX+10] Ding YUAN, Haohui MAI, Weiwei XIONG, Lin TAN, Yuanyuan ZHOU, et Shankar PASUPATH. « SherLog: Error Diagnosis by Connecting Clues from Run-time Logs ». In *ASPLOS'10: Proceedings of the 2010 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, Pittsburgh, PA, USA, 2010.

[YPH+12] Ding YUAN, Soyeon PARK, Peng HUANG, Yang LIU, Michael M. LEE, Xiaoming TANG, Yuanyuan ZHOU, et Stefan SAVAGE. « Be conservative: enhancing failure diagnosis with proactive logging ». In *OSDI'12: Proceedings of the 10th USENIX con-*

ference on Operating Systems Design and Implementation, pages 293–306, Hollywood, CA, USA, 2012.

[YZP+] Ding YUAN, Jing ZHENG, Soyeon PARK, Yuanyuan ZHOU, et Stefan SAVAGE. « Improving software diagnosability via log enhancement ». In *ASPLOS'11: Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–14.

[ZCA+06] Feng ZHOU, Jeremy CONDIT, Zachary ANDERSON, Ilya BAGRAK, Rob ENNALS, Matthew HARREN, George NECULA, et Eric BREWER. « SafeDrive: Safe and recoverable extensions using language-based techniques ». In *OSDI'06: Proceedings of the 2006 Symposium on Operating System Design & Implementation*, pages 45–60, Seattle, Washington, 2006.