A Thesis for the Degree of Ph.D. in Engineering

A Study on Faults and Error Propagation in the Linux Operating System

March 2016

Graduate School of Science and Technology
Keio University

Takeshi Yoshimura

Acknowledgement

I would like to thank my adviser, Prof. Kenji Kono. His guidance helped me in all the time of research. I would like to express my sincere gratitude to Prof. Hiroshi Yamada. This dissertation would not have been possible without their advice and encouragement.

I am also grateful to the members of my thesis committee: Prof. Shingo Takada, Prof. Hiroaki Saito, and Prof. Kenichi Kourai. This dissertation was greatly improved by their invaluable feedback.

During my Ph.D., I did an internship at NEC. I enjoyed working with Dr. Masato Asahara and the opportunity had a significant impact on my research skills. He also taught me LDA, which is the core of the fault study in this dissertation.

I am also thankful to my colleagues in the *sslab*. Their surprising enthusiasm and skills have always inspired me.

I appreciate the financial supports from the Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists and the Core Research for Evolutional Science and Technology of Japan Science and Technology Agency.

Finally, I would like to thank my family, my parents, sister for their support all these years. Without their support and encouragement, many accomplishments in my life including this dissertation would not have been possible.

Abstract

A Study on Faults and Error Propagation in the Linux Operating System

Takeshi Yoshimura

Operating systems are crucial for application reliability. Applications running on an operating system cannot run correctly if the operating system fails. Recent studies reveal that naive faults such as NULL pointer dereferences are still prevalent in the Linux operating system, which is widely used in productions such as Android smartphones, cloud platforms, and air traffic control systems. Existing approaches to prevent operating system failures are twofold: fault detection and failure recovery. Fault detection is the approach to find and fix as many faults as possible before shipping, using techniques such as static analysis and software tests. Failure recovery is the approach to tolerating or mitigating the failures caused by undetected faults, using the techniques such as software rejuvenation.

To advance the state of the art of fault detection and failure recovery for operating systems, this dissertation conducts detailed analysis of faults and error propagations in the Linux operating system. Many efforts have been devoted to improving the quality of fault detection and failure recovery for operating systems. However, existing techniques rely on ad hoc intuitions and experiences of operating system developers without understanding the overall trends in Linux faults and error propagations. For example, if the developers notice that there are many faults in which NULL pointer check is missing, a static code checker is developed

to check for missing NULL checks. Failure recovery is pessimistic and assumes the entire kernel is always corrupted by a single error.

To understand faults in Linux, this dissertation analyzes more than 370,000 Linux patches, code modification records for Linux in English. To extract topics in the patches, Latent Dirichlet Allocation, a technique of natural language processing is applied and the patches are classified into 66 clusters based on the extracted topics. To demonstrate the resulting clusters can contain useful information to develop sophisticated code checkers, one cluster is deeply investigated and 160 patches for fixing faults related to interrupt handling are extracted. Based on the knowledge obtained from the extracted patches, a static code analyzer has been developed and detected five unknown faults in Linux 4.1.

This dissertation also investigates error propagations in Linux. To analyze error propagations, a new concept called error propagation scope is proposed. Error propagation scope specifies how far an error can propagate. In the dissertation, two scopes, process-local and kernel-global errors, are introduced. Process-local errors do not propagate beyond process contexts inside the kernel. Kernel-global errors corrupt shared data structures and propagate beyond process contexts. This dissertation shows 73% of errors are process-local and do not propagate beyond the in-kernel process contexts in the experiments. This result indicates that there are chances to avoid kernel crashes in Linux by killing a failing process.

The contribution of this dissertation is twofold. First, an analysis of fault reports helps develop new static code checkers that can detect faults overlooked in an ad hoc approach. Second, partial recovery of the Linux operating system deserves further investigation and research to achieve lightweight countermeasures for system failures.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Operating systems are crucial for computer systems reliability. Operating system kernels fail less frequently than applications, but failures on them cause a severe impact on the entire software stacks. Even if applications running on an operating system are highly available, a kernel fault results in a failure of the applications. It may lead to a serious service outage, security vulnerability, or performance degradation. Any failures on computer systems are never negligible because they are the basis of modern internet services. For example, the estimated cost of Amazon Web Service outages in 2013 was $1,104 per second on average [64].

Linux is a widely used operating system in products that are essential for daily life. In 2015, Android smartphones had 1.4 billion active users [93]. As well as in personal devices, Linux is used in 97% of Web front-end servers [96]. In the Amazon cloud, there are more than 200 thousand instances of Linux-based operating systems [94]. Recently, mission-critical systems also run on Linux to reduce development costs. In Germany, an air traffic control system runs on top of Linux [35].

However, as shown by Palix et al., faults in Linux are not eliminated for a decade [76]. The study shows that new features in Linux introduced new faults while developers fixed many faults. As a matter of fact, file systems in the Linux kernel is evolving and introducing faults [62] although they seem more stable than other kernel components. Moreover, according to a study of reports in Com-

mon Vulnerabilities and Exposures [19], kernel developers cannot avoid introducing known security problems such as arbitrary memory corruption and denial-of-service. In Amazon's recommend virtual machine instance, 40 updates for kernel vulnerabilities have been published since it was born in September 2011 [1].

Throughout this dissertation, software bugs are described by three terminologies: *fault*, *error*, and *failure* [77]. A *fault* is a defect in a component. This work focuses on defects that are derived from programming mistakes in the C language code of the Linux kernel. "*Inject* a fault" means introducing a defect artificially in a component. "*Activate* a fault" means that a fault becomes effective; a faulty region in a component is executed. An *error* is a state that is changed by a defect. An error is caused by a fault activation. An error often *propagates* from one component to another, thereby creating new errors. "*Manifest* an error" means that a defect affects a delivered service by triggering a *failure*, which is deviant behavior (e.g., a kernel crash).

1.1 Preventing Failures in Operating Systems

Existing approaches to prevent operating system failures are twofold: fault detections and failure recovery. Many efforts have been devoted to improving the quality of the two approaches.

1.1.1 Fault Detection

Fault detection is the approach to find and fix as many faults as possible before shipping. This dissertation classifies the techniques into static analysis, software testing, formal proof, and system diagnosis.

Static analysis enables developers to check typical faults in large-scale code. In particular, this technology easily validates execution flows in exceptional conditions, which developers often overlook. Much work uses static analysis to find various kinds of faults in Linux: resource-release omissions [86], integer overflows [97], hardware failure mishandling [48], and so on.

Software testing enables developers to confirm that target software runs correctly under given test cases. The advantage of testing is to validate dynamically invoked code that static analysis cannot easily track. For example, concurrency faults [63] in Linux are detected by a specialized virtual machine monitor that enforces kernel interleavings [36]. Another example is to validate the virtual CPU implementation [2].

Formal proofs verify the satisfaction of given specifications including no semantic violations. seL4 [51] guarantees no buffer overruns and dangling pointer accesses, which cause serious security vulnerabilities. In Linux, there are efforts of proving the functional correctness of BSD packet filtering [98] and file system robustness [20].

System diagnosis with crash dumps and runtime logs is necessary for fixing broad ranges of complex faults. In the case of Microsoft, developers found bugs in Windows NT and Office that had existed for over five years through their large-scale analysis of crash dumps [38].

1.1.2 Failure Recovery

Failure recovery is the approach to tolerating or mitigating the failures caused by undetected faults. In this dissertation, techniques for failure recovery is classified into fault tolerance, software rejuvenation, and fault isolation[1].

Fault tolerance mechanisms can undo erroneous actions without reboots. It is derived from the requirement for tolerating hardware faults in software. However, it is also applied for mechanisms to recover systems from transient software errors [90] [57] [47]. Transient errors can happen in systems by concurrency faults, timing dependent behaviors of devices, and so on. A primary challenge of fault tolerance is large runtime overheads of execution logging to undo erroneous actions afterward.

Software rejuvenation is a lightweight technique to prevent software aging-related failures. Aging-related failures in Linux mostly happen by memory

[1]The terminology "fault" is used for the names although they do not remove faults in software as Chapter 2 describes in detail. Thus, this dissertation classifies them as failure recovery.

leaks [23], which cause high memory pressures on systems that run for a long time. To prevent aging-related failures in operating systems, fast reboot mechanisms are effective to reset the accumulative errors in a system without long downtime [30] [104].

Fault isolations logically separate software domains and cure errors by restarting a faulty domain. For example, a dedicated address space for device drivers [91] [44] will avoid errors due to buffer overflows and dangling pointer accesses in a device driver. Ensuring API[2] integrity [67] prevents error propagation at the interface between device drivers and the kernel core. Kuznetsov et al. introduce the concept of code-pointer integrity from the fact that many security vulnerabilities exploit arbitrary memory writes to craft function pointers [54].

1.2 Motivation

Section 1.1 implies that each technique to prevent operating system failures focuses on particular types of faults. However, faults are derived from human mistakes, and thus, it is difficult for developers to predict what kinds of faults they should target.

As a result, existing techniques rely on ad hoc intuitions and experiences of developers without understanding the overall trends in Linux faults and error propagations. For example, if the developers notice that there are many faults in which NULL pointer check is missing, a static code checker is developed to check for missing NULL checks. Failure recovery is pessimistic and assumes the entire kernel is always corrupted by a single error.

Unfortunately, it is not an easy task to obtain deep experience of target systems. In open-source projects such as Linux, there are many contributors, but no formal means to share experiences. In addition, a large-scale software system consists of many components, each of which requires different expertise to develop mechanisms for fault detection and failure recoveries such as static analysis and

[2]In operating system research, API (Application Programming Interface) often refers to in-kernel functions that are exposed to third-party device drivers and file systems

software rejuvenation.

1.3 Study Overview

This dissertation conducts detailed analysis of faults and error propagations in the Linux operating system. The result leads to some insights to advance the state of the art of fault detection and failure recovery for operating systems.

To understand faults in Linux, this dissertation analyzes 370,000 and more Linux patches, code modification records for Linux in English. The analysis uses natural language processing and machine learning to extract fault patterns whose occurrence is statistically frequent in the fault-fix repositories. Using a technique of natural language processing, Latent Dirichlet Allocation (LDA) [10], the fault reports in the repositories are classified based on topic. Closely related fault reports are expected to contain similar fault patterns.

As a result of the clustering, the patches are classified into 66 clusters. To demonstrate the resulting clusters can contain useful information to develop sophisticated code checkers, one cluster is deeply investigated and 160 patches for fixing faults related to interrupt handling are extracted. Based on the knowledge obtained from the extracted patches, a static code analyzer has been developed and detected five unknown faults in Linux 4.1.

This dissertation also investigates Linux behaviors under error propagations. To analyze error propagation, This dissertation introduces a concept of error propagation *scope*. Error propagation scope specifies how far an error can propagate. The error propagation scope is *process-local* if an error is confined in the process context that activated it. The scope is *kernel-global* if an error propagates to other processes' contexts or global data structures.

The experimental result shows that 73% of errors are process-local and do not propagate beyond the in-kernel process contexts. This result indicates that Linux could continue to run safely in 73% of failures if a failing process is killed. Since an error was not propagated to other process contexts, the kernel could be recovered to a consistent state simply by revoking the context of the faulty process.

The contribution of this dissertation is the following:

- A large-scale analysis of fault reports helps develop new static code checkers that can detect faults overlooked in an ad hoc approach. Concretely, five fixes that the static analysis found were accepted by Linux maintainers and landed on the upstream kernel. All the bugs the static analysis found have existed for three to ten years in Linux, although it is one of the most frequently-used software in the world for decades. The result implies the utilization of software repositories for static analysis is promising to enhance the future software quality by detecting bugs that many developers overlook.

- Partial recovery of the Linux operating system deserves further investigation and research. According to the study of error propagation scope, there are chances to avoid kernel crashes in Linux. The detailed analysis of errors shows that global error propagation is prevented by frequent defensive coding in the Linux kernel.

1.4 Organization

This dissertation is organized as follows. Chapter 2 describes existing techniques to prevent failures of operating systems in detail. Discussions in the chapter motivate the field study of faults and error propagation in this dissertation. Chapter 3 overviews a large number of field studies that are conducted to lead to new insights into operating systems developments. Studies in this dissertation can be regarded as a field study of operating systems like existing ones. Chapter 4 shows the result of using natural language processing to study faults in Linux. The chapter also includes the detail of a demonstration to use the extracted knowledge to develop static analysis. Chapter 5 describes the introduction of error propagation scope and the results of studying errors in Linux using fault injections. The chapter additionally shows an in-depth analysis of error propagation in Linux from the view of the C-language code. Chapter 6 concludes this dissertation and discusses the

future directions.

Chapter 2

Failure Preventions for Operating Systems

A primary topic of operating systems research is countermeasures for faults and errors in operating systems. They tackle wide varieties of issues of low availability, security vulnerabilities, and performance degradations. Despite the diversity of solved issues, their approaches are twofold: fault detection and failure recovery as shown in Chapter 1. This chapter briefly overviews these existing work to motivate studies in this dissertation.

2.1 Fault detection

2.1.1 Static Analysis

Static analysis helps developers check if given implementations match to particular code patterns. The primary advantage of static analysis is the scalability to large-scale code like Linux due to its fully-automated checking. Furthermore, static analysis is useful to check code in exception error handling, which is difficult for software testing to check.

Engler et al. introduce checking implicit system rules in the code of operating system kernels with their meta-level compiler [33]. The work enables developers

to define various rules such as "Releasing locks after acquiring them" by hand.

Since then, numerous work proposes to check various code patterns for detecting faults specific to operating systems. Collateral evolutions [74] [75] is API usage changes that often cause developers' mistakes. Their idea is derived from the unstable property of Linux in-kernel APIs [53]. Carburizer [48] checks code patterns that do not tolerate hardware failures. KINT [97] detects integer overflow, which potentially lead to security vulnerabilities of operating systems. STACK [99] searches ill-formed code that compilers inappropriately optimize. Unfortunately, it is unclear how developers obtain new patterns like the above efforts.

In fact, an obstacle to finding faults by static analysis remains how to know what to check. A concept of "specification mining" aims to extract particular patterns from given code by using techniques such as statistical methods [34] [60]. However, they still cannot avoid a primary limitation such as false positives and negatives of checked patterns due to the characteristics of static analysis. Hector [86] focuses on resource-release omissions to reduce a large number of the false patterns. However, they cannot extract code patterns involving dynamically called code sequences, which are frequently contained in operating systems. Yang et al. [105] extract file system specifications from a source code by tracking file system behavior, although they do not answer if the idea can be applied for more complex interactions between other types of drivers and devices.

Consequently, existing techniques for static analysis often rely on ad hoc OS developers' experiences and intuitions. An example of static analysis accepted by Linux developers is Coccinelle [56]. It extracts Linux API protocols by utilizing the insights of experienced developers. In the case of Windows, Static Driver Verifier (SDV) [5] focuses on Windows stable in-kernel APIs to detect faults in Windows device drivers. SDVRP [7] enables developers to apply the SDV concept for other software with a more robust and efficient analysis engine (SLAM2) [6] [87].

However, especially in Linux, it remains unclear if there can be other faults that developers should check during their development. Currently, DDVer-

ify [102] [101] [29] verifies 1642 properties of 31 device drivers in Linux 2.6.16 [101]. Linux Driver Verifier [110] verifies more than 40 rules of 3,300 drivers in Linux. To determine faults specific to operating systems, developers need to obtain the overall trend of faults in operating systems like efforts in distributed systems [106]. They investigate a sufficient number of catastrophic failures like data losses in distributed file systems and found the effectiveness of checking exception handling.

2.1.2 Software Testing

Software testing reduces developers' debugging efforts as well as static analysis. It automates the process of ensuring the functional correctness of software under given test cases. An advantage of testing is the ease of validating much functional correctness in no matter how complex and large-scale software is. In particular, static analysis cannot conduct join testing involving hardware unlike the validation by software testing.

An example of complex functionalities that existing work validated is CPU virtualization [2] because they involve hardware-specific behaviors with multi-tier software stacks. Another example is concurrent behaviors [36] because these faults are activated in very narrow time windows. However, the biggest challenge of software testing is to obtain test cases that validate as many software behaviors (including exceptional ones) as possible.

A primary solution to obtaining test cases is to utilize symbolic executions of static analysis. Symbolic executions conceptually fork all possible paths in a code, calculating and preserving all variable constraints in contexts. Klee [13] utilizes the property to automatically find inputs that cause failures. SymDrive [82] running on top of S2E [21] introduces symbolic devices to check driver behaviors with various device models. However, exceptional behaviors that test cases should often cover happen in deep inter-procedural flows, and thus, the problem of path explosions arises. Deep inter-procedural analysis mostly suffers from memory pressures and heavy scheduling of a large number of contexts. The work for software testing using symbolic executions has to give up scheduling low-

priority contexts and reduce the number of the parallelism. However, it may not be a primary limitation because distributed symbolic executions can mitigate the impact [12]. Unfortunately, there remains the limitation of static analysis as Section 2.1.1 discusses.

An alternative approach is to artificially reproduce exceptional behaviors using fault injections. LFI [68] emulates failures inside user libraries to validate error handling in applications. LFI may need to be extended to validate using tightly-coupled in-kernel libraries since LFI focuses on well-defined interfaces of user libraries. G-SWFIT [31] [24] and FINE[50] mutate binary instructions to emulate general faults in software or hardware. SAFE [71] focuses on open source code to emulate general software faults to improve the accuracy of fault injections. CloudVal [78] is a framework to inject faults into operating systems under virtualized environments. Faults specific to operating systems can be emulated by the means of fault injection techniques here. However, it requires appropriate fault characteristics to be emulated.

In real-world software developments, testing tends to be ad hoc. For the example of LLVM [55] development, it offers every patch submission to attach test scripts [81]. Test scripts are often simple C language code that reproduce failures of the compiler without a submitted patch. The scripts are accumulated in the public repository as well as the main source code and directly used as test suites that are included in the development package. In the case of Linux, there is a public testing project [80] although it is currently separated from the kernel main developments. Although this dissertation does not focus on software testing, understanding faults in the wild potentially leads to finding overlooked test cases with the "bird's-eye" view.

2.1.3 Formal Proof

The theorem proof of programs with Hoare logic directly guarantees the functional correctness of target software under the given specifications. The obstacle to applying formal proofs for operating systems is type safety and determinism, which are main assumptions for the logic. Operating systems in productions like

Linux and Windows are developed with the C language, which is difficult to preserve the arbitrary type safety. Moreover, operating systems parallelize many resource usages with interrupts and polling in various components to avoid low resource utilization.

seL4 [51] [32] [9] solves the two issues with Haskell prototyping and verification friendly architecture. However, the work reveals that theorem proofs require person-year-long efforts whenever the specification is modified although the process of coding and proving is semi-automated. Besides, a fully-verified kernel is difficult to be achieved without modifying fundamental operating system designs such as interrupts and memory managements. The partial verifications in Linux are performed on particular features: a file system [20] and BSD packet filtering [98].

As a deviant of formal proofs, there are attempts to automatic generation of kernel code by formal specifications. Formal specifications enable third-party developers to (semi-)automate the device driver development without the extra expertise of operating system internals and potentially avoid faults that developers make [84] [85]. Unfortunately, existing work of formal specifications supports limited device types and features.

Deductive verification efforts in practice are incomplete because there are many faults in the Linux kernel core as shown in a study of vulnerability reports [19]. Besides, the correctness of specifications, which every proof assume, requires appropriate knowledge of what code is unpreferable. For example, developers know that buffer overflows should be avoided while there can be uncovered semantics that developers should verify as shown in recent studies in rules of packet filtering [18].

2.1.4 System Diagnosis

Runtime logs and dumps help developers diagnose systems and often result in fixing wide varieties of faults. They include even crashes involving complex dependencies of software stacks and performance degradations. However, primary issues on system diagnosis are to reduce developers' efforts by automating the

process.

Logs enable developers to obtain runtime information that developers can customize to find root causes easily. SherLog [107] automates the process of narrowing down failed control-flows from log messages. However, SherLog assumes all the logs have sufficient information at appropriate places to track control-flows. LogEnhancer [109] automatically re-writes existing logs to contain sufficient contents so that developers can trace failed control-flows. Errlog [108] detects error paths and insert logs in the path. However, the above techniques inherently assume static analysis, which have limitations to solve complex problems in code as shown in Section 2.1.1.

Another issue of system diagnosis is to reduce efforts to understand large amounts of data. Xu et al. [103] use machine learning to extract anomaly information from millions of lines of console logs. Windows Error Reporting prioritizes billions of Windows crash dumps with clustering and heuristics [38] [27].

A problem specific to analyze Linux crash dumps is the diversity of kernel images. Guo et al. mitigate the issue by extracting the line of C language code that incurs a crash from binary-level information [42]. In Linux, crash dumps are emitted through a crash procedure that is called kernel oops. The procedure is very naive but lightweight compared to Windows bluescreen event. The concept of error propagation scope that this dissertation introduces is derived from the property of killing a faulty process at the kernel oops. Thus, the study in this dissertation can be regarded as the evaluation of the reliability of crash procedure in Linux. In other words, the error study in this dissertation also may lead to efficient collections of crash dumps to advance post-mortem of operating systems.

2.2 Failure Recovery and Mitigation

2.2.1 Fault Tolerance

Failure recovery and mitigations are derived from the requirements of highly available systems with fault-tolerant mechanisms. Fault tolerance recovers sys-

tems from errors before failures happen. A primary method of fault tolerance in operating systems is the additional execution monitor to roll back them. Conventional fault-tolerance by redundant hardware is usually not preferable regarding the cost-effectiveness and strong constraints of available environments.

Shadow driver [90] intercepts and records device inputs and perform replaying records to recover from transient failures involving device behaviors. However, shadow driver cannot be applied to file systems because their states are not transient. Membrane [89] achieves restartable file systems by checkpointing of persistent states on disk platters. However, Membrane focuses on file systems, and thus, also assumes characteristics of particular kernel components. Recovery Domain [57] generalizes the concept by separating general kernel contexts into protection domains that can be undone. In virtualized environments, an adapted virtual machine monitor can simplify the mechanisms of interpositions. Remus [26] intercepts all the inputs of guest OSes at the virtual machine monitor layer to asynchronously replicate their states.

However, these techniques cause high runtime overheads. FGFT [47] focuses on protecting a single driver that users specify with fine-grained checkpointing and a speedy fault isolation [70]. However, it requires developers' efforts to identify and specify the isolation area in device drivers. Tardigrade [61] replicates only lightweight applications built with a Library operating system for Windows [79] by monitoring library interfaces [8]. However, it assumes virtualized environments and services can be built with the library. As a result, existing fault tolerant mechanisms are often difficult to be applied for many environments such as smartphones.

In the real world, there have been attempts at creating operating system kernels capable of failure recovery. Multics and MVS [4] are known products of recoverable kernels, although the recovery code dominates half of the operating system code. Unfortunately, even these systems cannot avoid critical faults and outages that require reboots [88].

2.2.2 Software Rejuvenation

Software rejuvenation [45] is a concept that re-uses initializations of software modules to error recoveries. Using re-initializations for error recoveries enables systems to avoid complex routines for recovery events that rarely happen but often become huge runtime overheads. This technique assumes that operating system failures are often transient. Transient failures are caused by timing dependent concurrency faults, accumulated memory leaks, and other faults that are difficult for developers to fix before shipping.

An issue of software rejuvenation is to determine fine-grained modules for high efficiency. Microreboot [15] increases service availability by fine-grained modules and reboots. The work also shows the required properties of microrebooting such as crash-only property [14].

For operating systems, the simplest approach of software rejuvenation is kernel-level reboots. Otherworld [30] gives applications opportunities to survive operating system failures. The technique omits the time of hardware resets and service downtime. However, applications need to be added extra, application-specific crash procedures so that the applications can preserve application-specific states during kernel rebooting. Phase-based reboot [104] shortens downtime by utilizing virtual machine snapshots. Both of the two techniques are based on the pessimistic idea to error propagations; they always try to recover systems from extreme situations where a single error causes the entire system corruption.

2.2.3 Fault Isolation

Numerous proposals in software fault isolation for operating systems often focus on isolating kernel extensions such as device drivers and file systems. They try driver-level reboots (reinitializations of kernel extensions) to recover systems from errors detected by isolation mechanisms. As well as kernel-level software rejuvenation, driver-level reboots assume that a single error causes the entire driver corruption.

Microkernel architectures such as MINIX3 [44], seL4 [51], and CuriOS [28]

ensures fault isolations of each server in the architecture-level. In Linux, Nooks [91] separates address spaces for device drivers to prevent buffer overflows and preserve in-kernel API semantics. However, Nooks require additional modifications of device drivers. SUD [11] enables user-level drivers to access privileged functionalities such as direct memory access (DMA) and interrupts with few driver modifications. Recently, security concerns in hypervisors arise due to the widespread use of cloud computing. Hyperlock [100] ensures that each guest operating system runs on top of a dedicated KVM module to protect host operating systems from being exploited by a crafted guest.

These existing work reveals that fine-grained isolations by microkernel-like architectures often cause high latency or low throughput by added complex interfaces for safe message passing. To reduce the runtime overhead, optimizations by compile-time instrumentations are proposed [111] [16] [67]. However, they additionally require developers' efforts to write in-kernel API wrappers or annotations, which are necessary for accurate isolations.

2.3 Summary

Existing techniques for failure preventions lack the view of the characteristics of faults and errors. As a result, they tend to rely on ad hoc developers' experiences and intuitions.

As shown in Section 2.1.1, static analysis is useful to validate software thoroughly. A limitation of static analysis is the difficulty of validating complex code. However, Section 2.1.2 shows that software testing can cover the complex cases with appropriate test cases. Unfortunately, advanced software testing also relies on static analysis and does not solve the primary challenge to obtaining what to check. Section 2.1.3 implies that formal proofs may be promising to ensure the quality of code, but there remain strong limitations to apply for Linux, currently.

Section 2.2.1 shows that failure recovery is derived from the notion of fault-tolerance and explored to reduce runtime overheads. However, the efforts often focus on particular environments regardless of modern environment diversity. Ex-

isting software rejuvenations for operating systems are pessimistic and assume the entire kernel is always corrupted by a single error. As discussed in Section 2.2.2, coarse-grained modules may cause unnecessary recovery of states. However, exceedingly fine-grained isolations may cause complex dependencies and runtime overheads as shown in Section 2.2.3. Moreover, determining fine-grained isolations requires deep knowledge and experiences of system implementations as well as the issue of static analysis.

Chapter 3

Related Work

Decades ago, failure analysis of Tandem operating system [39] revealed that software faults are the biggest cause of the system failures. Since then, numerous studies of software faults and errors have been conducted. This chapter overviews existing studies to discuss the importance of this dissertation.

3.1 Faults in Operating Systems

Field studies of faults are useful to enhance existing fault detection techniques as Chapter 4 shows. These studies are usually performed with past failure logs and bug reports for target systems. The fault study in this dissertation is also investigating the fault reports in the Linux GIT repository.

Lu et al. study over 5,000 patches in Linux 2.6 file systems [62]. They show file systems evolution including the distribution of typical software faults. The file systems study gives the first comprehensive view of file system faults in the wild. However, the investigation is limited to particular file systems and difficult to scale to other components because the study is conducted manually. The issue can be solved by using static analysis to study faults [76] [22] while they cannot capture all types of faults as shown in Section 2.1.1. The study of faults in this dissertation avoids the issue with the help of natural language processing.

The file systems study [62] shows that concurrency faults are frequent in Linux

file systems. Most of the code in Linux can be invoked concurrently, and thus, careless code can easily cause atomicity violations, deadlocks, and so on. The detailed classes of concurrency faults are defined and studied by Lu et al. [63]. The fault study in this dissertation focuses on faults that potentially cause atomicity violations in device drivers, which the file systems study [62] does not cover.

An analysis of CVE reports shows a taxonomy of Linux faults related to security vulnerabilities [19]. It includes memory-related faults such as buffer overflows and dangling pointer accesses, which appear in the file systems study [62]. The result of the clustering in this dissertation does not contain security vulnerabilities as a representative cluster. This dissertation focuses on frequent patterns, and thus, do not extract such rare but serious faults.

Numerous work for static analysis shown in Section 2.1.1 can be regarded as studies of particular fault types. These studies indicate detailed and focused views of particular faults instead of comprehensive views of target software. There are studies of resource-release omissions [86], integer overflows [97], hardware failure mishandling [48], and so on. However, it is unclear how the knowledge of these fault patterns is obtained. This dissertation extracts the knowledge from a large number of fault reports.

Field studies of faults usually suffer from a large amount of noise in the target resources. Information retrieval and/or machine learning is useful for identifying fault fixes [92] and de-duplicating fault reports [83] [46]. They focus on extracting a particular collection of fault reports by analyzing the textual characteristics of the reports. In this dissertation, the study of faults analyzes textual data in fault reports as well as their work. However, the study focuses on extracting the content of fault descriptions using latent Dirichlet allocation [10].

3.2 Hardware Faults

Kadav et al. show faults of handling hardware failures in device drivers [48]. For example, many device drivers mistakenly assume that devices always output correct values. Such wrong assumptions potentially cause system hangups by infinite

loops in device drivers. The study implies that an important role of operating system is to detect hardware faults and display it to users. Thus, it is essential for operating system developers to know characteristics of hardware failures.

The largest-scale study of hardware failures is an empirical study of millions of Windows crash dumps [73]. They focus on personal computers, which often consist of commodity hardware. The work shows that the recurrent property of hardware failures: a machine that encountered hardware failures cause the second one intermittently. Ganapathi et al. also analyze crash dumps of Windows XP to show failures of device drivers [37]. However, they do not provide details of error propagation because crash dumps do not directly show fault information. This dissertation does not use crash dumps in the study of error propagation, unlike these two analysis.

The empirical study [73] also shows that DRAM is sometimes unreliable. For example, cosmic rays cause transient bit-flips on DRAM chips [112]. Li et al. show that non-transient or persistent errors on DRAM chips cause impacts on software stacks as well as transient errors [59]. In this dissertation, hardware faults are out of scope but bit-flips on DRAM may cause similar faults that the error study in this dissertation focuses on. In other words, recovering software failures may lead to tolerating transient faults in DRAM.

3.3 Operating System Behaviors Under Errors

Understanding the operating system behaviors under errors can be an aid for kernel developers to improve the kernel dependability or develop the mechanisms for kernel recoveries. However, real faults in operating systems are difficult to reproduce in experiments. Consequently, existing study of error propagation in Linux is investigated by using fault reports or fault injection as well as studies in this dissertation.

Gu et al. use fault injection to characterize Linux behaviors under errors [41]. Their analysis shows that crash latencies are often within ten cycles and also shows how an error propagates between Linux subsystems. Chen et al. [17] and

another paper from Gu et al. [40] investigate behavioral differences caused by different combinations of CPU models and operating systems. However, the fault models considered in these studies are device-level transient faults such as DRAM bit-flips. The study of error propagation focuses on low- and high-level programming mistakes.

Kouwe et al. automate the study of *fail silence violations* by comparing normal and faulty executions of an operating system [52]. Fail silence violations are not obvious crashes but prevent applications from continuing their execution. Unfortunately, they focus on external anomalies, i.e., do not conduct in-depth analysis of error propagations inside operating systems. This dissertation shows the detail of error propagation that cause fail silence violations in the view of C language code.

Cotroneo et al. analyze software aging in the Linux kernel with reports in Bugzilla [23]. They also investigate software aging in file systems [25]. Software aging is an error type that requires a long period for error manifestations. However, the error study in this dissertation tracks every kernel executions, and thus, this dissertation does not show the characteristics of error propagation that such long-running workloads cause. To cover these failures, this dissertation also investigates not-manifested errors and show the reason the errors do not result in failures.

3.4 Environment Evolutions Around Operating Systems

Lu et al. show that file systems in Linux are still evolving although the evolution causes software faults [62]. Their result implies that the environment evolutions that surround operating systems cannot be ignored when discussing operating system reliability.

For example, storages are turning into flash drives due to their performance superiority compared to conventional hard disk drives. However, a large-scale study of Facebook flash storages [69] shows uncovered limitations of flash drives. They

include the fact that the failure rate of flash storages increases at high temperatures, which are correlated with device power consumptions. As a result of this study, future device drivers potentially change the current policy of device power managements to reduce failure rates of flash drives. On the other hand, an in-depth analysis of faults in this dissertation shows that faults in interrupt handling also appear in the code for device suspends and resumes. The results imply that it will be important to check power management code as this dissertation does because the code may become more complex.

In addition to storage movements, application behaviors evolve as Harter et al. shows [43]. The study reveals that applications for multimedia and productivity do not access files sequentially although operating systems assume that file accesses are often sequential. On the other hand, the trend changes by applications are not observed in the study of faults in this dissertation. This is because the objective of the study is to overview the patches in the past; such recent trends may appear if the given data set is changed into patches within recent periods.

As well as storages, modern devices are evolving. Kadav et al. study the characteristics of modern devices and show the future direction of improving device drivers reliability [49]. For example, they investigate the interactions between devices and drivers. It results that some modern device classes and buses such as USB are loosely-decoupled with each other by well-defined interfaces. The authors expect that the interfaces are useful for future fault isolations. However, for current Linux device drivers, the fault study in Section 4 also implies that such interactions may be error-prone due to the increases of software complexity.

3.5 Summary

According to Section 3.4, operating systems still need to evolve although the evolution causes new faults. As a result, there is numerous work to study software faults and errors in operating systems. Unfortunately, these studies still cover a limited portion of faults in operating systems due to limitations of their methodologies.

Section 3.1 shows that operating systems also contain well-known faults such as buffer overflows and concurrency faults. Faults specific to operating systems are caused by mishandling hardware failures that Section 3.2 shows. However, studies of faults tend to focus on particular kernel components and fault patterns: they lack top-down views of software faults in operating systems. The limitation is derived from the difficulty of manual investigations on a large number of study resources (e.g., fault reports). The study of faults in this dissertation avoids the issue with the help of natural language processing on a large number of fault reports.

Errors in operating systems are mostly unclear due to a limited number of studies as shown in Section 3.3. In particular, the influences of *software* faults are less studied although numerous work attempts to solve software faults as shown in Chapter 2. This dissertation focuses on software errors as well as software faults.

Chapter 4

Faults in the Linux Operating System

The objective of this chapter is to understand typical faults with more than 370,000 fault fixes in Linux GIT repository. As shown in Chapter 3, the biggest challenge of field studies is that the size of target resources is too large for developers to read in full.

A simple solution is to limit the number of investigations by selecting random resources within a particular period. The drawback of this approach is that it cannot determine whether the contents of sampled resources are rare cases or not. Another method is to extract the resources that contain bug-like keywords that developers already know (e.g., "NULL" or "race"). However, developers do not always know such keywords, especially when they attempt to study faults related to operating system semantics.

The solution of this dissertation to the issue is to use natural language processing to extract fault patterns whose occurrence is statistically frequent in the repository. Additionally, this chapter shows an experience of developing checkers with the knowledge extracted from the fault reports.

4.1 Linux Patch Characteristics

Linux patches have abundant information about faults that a system experienced. Target resources in existing work [62] are also patch documents written in natural languages. In particular, useful information about faults is provided as a form of natural languages. This section briefly describes observations of Linux patch characteristics, which lead to natural language processing shown in the next section.

Figure 4.1 shows a typical Linux patch on August 13, 2013. The example shows a typical race condition whose cause was a lock missing (missing calling functions `genl_lock()` and `genl_unlock()`) when dumping generic netlink families. Function `genl_lock` wraps `mutex_lock()` that is a common synchronization API in Linux kernel. At the very least, the function can be speculated as a synchronization family from the name without knowing the detail of the implementation. In addition, it reports a crash by unprotected list addition/removal. The author indirectly explains that they can activate the error only after the second call of `ctrl_dumpfamily()`. Then, the fault is fixed by adding a lock and unlock on function `ctrl_dumpfamily()` in a .c file under `net/` directory, which is a part of network stack code in the Linux kernel. The author reports the fault can exist in old Linux kernels, and finally, the patch is signed off by related developers. The patch document consists of a title, two body paragraphs, the last paragraph for the signature, and code change. However, the second body paragraph and the last signature are not useful for the objective of this chapter.

The example shows that developers can obtain all the above information on a real fault in the Linux kernel by understanding the context of the explanation part written in natural languages. As the example shows, patch documents are well-written and easy to understand even if there are mentions about device or model-specific problems. This is because developers other than the patch authors almost always review patches in the development process of Linux kernel. However, developers still need to determine problems on some patches and speculate what problems the code change can remove. Note that this is not a difficult task in many

commit 58ad436fcf49810aa006016107f494c9ac9013db
Author: Johannes Berg <johannes.berg@intel.com>
Date: Tue Aug 13 09:04:05 2013 +0200

genetlink: fix family dump race

When dumping generic netlink families, only the first dump call is locked with genl_lock(), which protects the list of families, and thus subsequent calls can access the data without locking, racing against family addition/removal. This can cause a crash....

A similar bug was reported to me on an old kernel (3.4.47) but the exact scenario that happened there is no longer possible ...

Signed-off-by: Johannes Berg <johannes.berg@...>
...

```
diff a/net/netlink/genetlink.c...
ctrl_dumpfamily(...)
        struct net *net = sock_net(skb->sk);
...
+       bool need_locking = chains_to_skip||...;
+
+       if (need_locking)
+               genl_lock();
...
+       if (need_locking)
+               genl_unlock();
+
        return skb->len;
 }
```

Figure 4.1: An example patch

cases because documents in Linux kernel are often well-structured by paragraphs as the example shows.

4.2 Methodology

This section shows the method for extracting typical faults from a large number of patches in Linux kernel. To this end, the study in this chapter uses natural language processing and clustering for extracting common patches from a large number of patches. The key idea of the method in this chapter is based on the observation of patches in Linux, which was described in the previous section.

In short, patch clustering is performed through patch weighing with latent Dirichlet allocation (LDA) [10] and top-down clustering [66], with the weight calculated by LDA. Top-down clustering with LDA groups similar patches while ensuring that their contents frequently appear among all the target patches. Finally, I perform manual inspections on patches that the clustering automatically extract to understand Linux faults.

This dissertation uses fault reports from the Linux upstream git repository excluding merge commits. The target resource in this chapter is 370,403 patch descriptions from Linux 2.6.12-rc2 on April 2005 to Linux 3.12-rc5 on October 2013.

4.2.1 Natural Language Processing

The previous section showed that patches in natural language,, i.e., English has abundant information about the faults. However, English has many kinds of well-known noise such as variation of verbs and nouns. Thus, the analysis first stems and drops noisy words as well as common natural language processing [66].

"Stemming" means grouping together words that have the same meaning but different grammatical variations (e.g., 'leaks', 'leak', and 'leaking'). This dissertation uses the Porter's stemmer, which is the de facto standard for stemming English [66]. It utilizes suffix stripping based on a rule to conflate inflected words

to a root. Stemming does not always preserve the root as a valid word, so the results contain partially mutated words such as “memori” instead of “memory”.

Noise words are not only well-known “stopwords” (frequently appearing words in English documents such as ‘is’, ‘a’, ‘that’) but also decimal and hexadecimal numbers (e.g., ‘8’, ‘16’, ‘0xff’, ‘1e’), except for ones that are bonded to other letters e.g., “x86”. Since the information about the development process is not relevant to the objective of this chapter, The natural language processing in this chapter also gets rid of paragraphs that include “Signed-off-by:”, which always appears in the signature paragraph in Linux patch reports.

The Linux coding style is well organized, and function names frequently appear in fault descriptions for Linux patches. For example, `fat_alloc_inode()` and `ext3_alloc_inode()` are functions for an inode allocation in FAT and ext3. In this case, clustering should probably extract general inode allocation failures by regarding them as similar words. To do so, the natural language processing in this chapter uses non-alphabet and non-digit words as split tokens in addition to spaces. From the example functions, two sets of words are obtained, e.g., (“fat”, “alloc”, “inode”) and (“ext3”, “alloc”, “inode”). The example appearing in the previous section, i.e., wrapping function `genl_lock()` and `genl_unlock()` should be divided into (“genl”, “lock”, and “unlock”), so that LDA can weigh the example document as heavy as other documents with “`_lock()`” functions.

LDA weighs documents with abstract “topics” that occur in a collection of documents. For example, when a developer finds a bug, he or she describes it using words such as “bug”, “problem”, “failure”, “crash”, or similar. When several such words occur frequently within a document, such word occurrences may shape the document’s semantics or context. Furthermore, they may appear more often in documents that contain similar topics. LDA automates the process of learning the topics by recognizing the pattern of word co-occurrences.

LDA is one of the main topic models in natural language processing. In LDA, a document is assumed to be generated by multiple topics, and the topics are assumed to be generated by multiple words that appear in the collection of docu-

ments. LDA infers a topic with the frequency of word co-occurrences in a given document. For example, if "memory" and "leak" frequently appear in the same report, LDA assigns a topic that weighs these two words high and other words low. LDA then weighs the assigned topic for reports that contain "memory" and "leak". Documents can be translated into probability sets of topics that the clustering can use as the weight of each document. This chapter uses the Apache Mahout [65] implementation of LDA. The detail of LDA algorithms is available in the original research [10] and Mahout documentation [65].

4.2.2 Top-Down Clustering

LDA produces the probability distribution of topics for each document. However, it is difficult to understand the meaning of the probability values themselves. For example, suppose that LDA estimates two documents as 0.3 and 0.4 for the probability of generating a particular topic. In that case, developers cannot determine if the two documents are similar or not by only reading these values. Thus, an additional method is necessary to group relatively similar documents among all the documents. In other words, this chapter uses LDA as a dimensionality reduction for improving clustering results [10].

To do that, the study in this chapter uses top-down clustering, which is a hierarchical clustering in which the algorithm starts from one cluster containing all the data and divides the cluster into two recursively. The division stops when all clusters become small enough and understandable (currently, less than 5,000 patches). The content similarity is calculated by the squared Euclidean distance between two vectors. The implementation of the division part is 2-means (k-means given k = 2) on top of Hadoop MapReduce [95] due to its scalable and concurrent-friendly properties, suitable for the processing of large amounts of data.

The example in the previous section mentions that old kernels have similar issues, which are not interesting for the objective of this chapter. Many patches for Linux can contain such noisy paragraphs as references to Bugzilla's URL, error logs, oops call traces, device ID tables, and test scripts. They frequently appear in fault descriptions, and LDA might mistakenly weigh the phrases higher and

weaken the actual fault descriptions. To avoid this issue, the clustering divides fault descriptions into paragraphs (such noisy phrases in Linux often appear as a paragraph) and then merge their results when top-down clustering checks the convergence.

This merging strategy is just to regard patches in the same cluster as their employing paragraphs. For example, when a patch has three paragraphs, p1, p2, and p3, whose clusters are c1, c2, and c1, respectively, This study regards the clusters to which the patch belongs as c1 and c2 (i.e., the frequency of each cluster is ignored). As a result, if the cluster c1 is not interesting, the results for c1 can be ignored while the interesting cluster c2 can be extracted independently.

4.2.3 Parameter Tuning

Mahout LDA provides parameters for the number of topics, number of iterations, two smoothing parameters, and so on. When the number of topics is small, LDA tends to weigh unrelated words into a topic, which potentially leads to clusters that gather unrelated patches. A large number of topics can capture more detailed contexts of documents. Although too many topics may result in it being too complicated for humans to understand, top-down clustering is also useful to filter out trivial topics among the corpus.

Table 4.1 shows example results of LDA that is executed with different numbers of topics (100, 250, and 500). In 100 topics, LDA generates a topic that is composed of “null”, “local”, “d”, “g”, and so on. Documents that have this topic with high probability are expected to have topics for NULL, but there can be many noise. Small numbers of topics often result in many noisy words such as “d” and “g” in a single topic. Furthermore, some words such as “local” imply that the topic also contains uninteresting ones such as local variables.

In 250 topics, LDA generates a topic that is composed of “pointer”, “null”, “reserv”, and so on. With larger numbers of topics, LDA often allocates noisy words in other topics. However, the topic still weighs the word “reserv” higher than “derefef”. In 500 topics, LDA generates a topic that is “null”, “derefer”, “dereferencc”, and so on. The topic is expected to have topics for NULL derefer-

Table 4.1: Topics That LDA Generates with Different Numbers of Topics
This table describes topics that LDA generates with different numbers of topics. The upper row in each table lists words that compose the topic. From left, words generate a topic with higher probability as shown in the below row. For example, with 100 topics, LDA generates a topic that "null" composes with the highest probability, 0.14.

Num. of Topics	Words Representing a Topic					
100 topics	Word	null	local	d	g	r
	Prob.	0.14	0.077	0.058	0.052	0.037
250 topics	Word	pointer	null	reserv	derefer	dereferenc
	Prob.	0.37	0.34	0.12	0.072	0.0020
500 topics	Word	null	derefer	dereferenc	deref	upgrad
	Prob.	0.71	0.15	0.043	0.017	0.014

ences while the topic is generated by the word "upgrad" with lower probability.

As shown in the table, LDA detects a detailed topic for "null" in 250 and 500 topics, compared to one for 100 topics. This study executes LDA with various numbers of topics until topics are enough detailed as demonstrated in this example. Unfortunately, due to time constraints, this study could not investigate more than 500 topics although some topics (e.g., "fs" topic shown in Section 4.3) can be divided.

The top-down clustering also has parameters for the number of patches that should be gathered into a cluster, measurements of the similarity among patches, and so on. As well as LDA topics, a small number of patches in a cluster can capture more detailed and focused faults. However, it increases the difficulty of overviewing all of the faults, because of an explosive increase in the number of clusters.

To determine how many patches should be included in a cluster, this study explored a visualized dendrogram (Figure 4.2.3 shows a few examples), which the top-down clustering generates. Each cluster is divided by 2-means as described

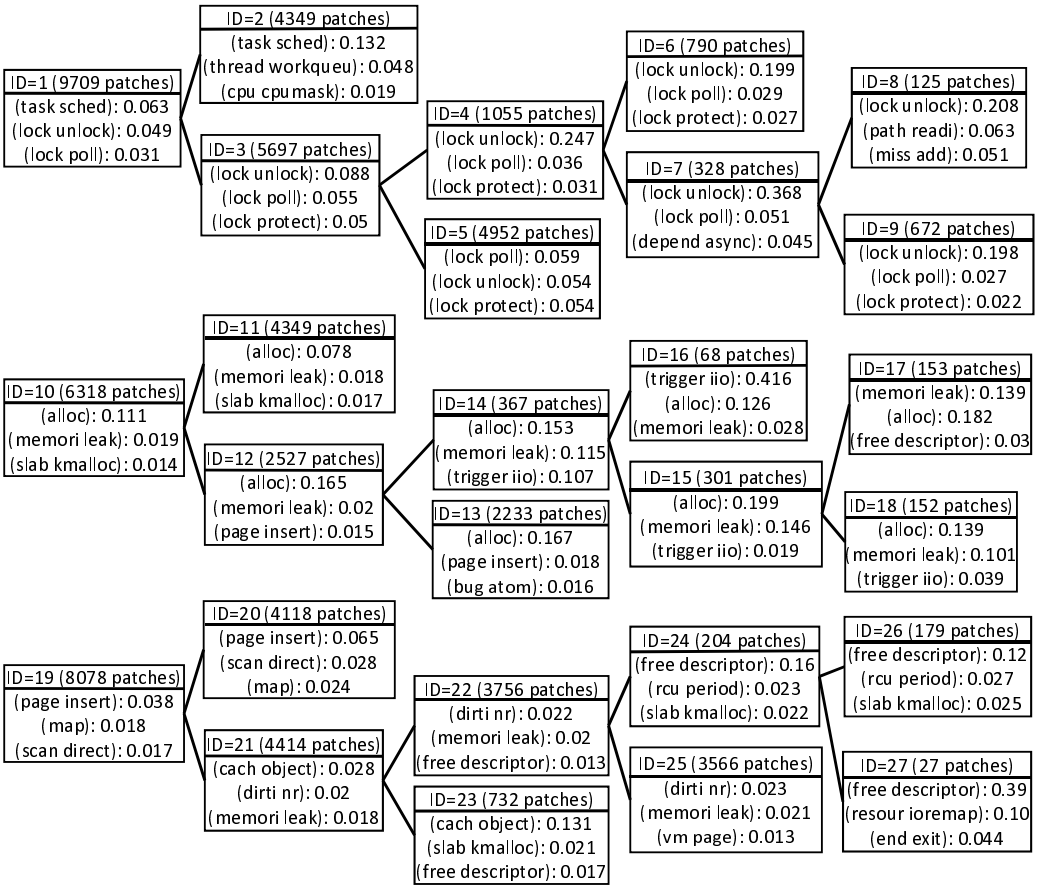

Figure 4.2: Example Dendrograms Generated by Top-down Clustering
Each rectangle shows a cluster that has a cluster ID, the number of patches, and the three most probable topics for the centroid of the cluster. Two edges of a rectangle show a parent-child relationships. Each topic is expressed by two most probable words of a topic (if highest probability is >0.9, expressed only by the word). Note that all the words are partially mutated by Porter's stemmer.

in Section 4.2.2. Note that the total numbers of patches in a pair of child clusters do not match ones in the parent cluster because of clustering with paragraphs. The figure shows that clusters with approximately 10,000 patches tend to include more than 1,000 unrelated patches. For example, the cluster ID=1 contained more

than 9,000 patches, but top-down clustering divided into two clusters that contained approximately 5,000 patches. The major topics for these centroids were also changed into scheduling and locking. Through the observation, this dissertation focuses on the results of clusters containing 5,000 to 10,000 patches.

Another known problem of LDA and top-down clustering is that they can fall into "local minimum" results depending on the initial result that they randomly generate. A workaround on this problem is to run them as many times as possible and explore better parameters. However, it is impossible to determine if or not the phenomena happens. Instead of tackling the issue, this dissertation indirectly shows the value of the clustering results by finding faults in Linux as described in Section 4.7.

This study takes numerous efforts towards parameter tuning as well as possible, but there might be better parameters. Parameter tuning requires a large number of computations with various combinations of parameters, and thus, time constraints become the primary problem for the accuracy of machine learning. In other words, the accuracy of this fault study can be improved by speeding up the methodology that efficiently distributes and parallelizes machine learning [58]. The proposed method in a research [3] also potentially improves the study accuracy by deciding better smoothing parameters in LDA.

4.3 Clustering Results

By extracting clusters containing 5,000 to 10,000 patches, LDA and top-down clustering resulted in 66 clusters.As discussed in Section 4.2.3, This study runs LDA with 1,000 iterations on 500 topics for inferring LDA parameters. Other hyper-parameters are given as suggested by Mahout's documentaters.

Although this study cannot give the absolute overview of Linux patches without inspecting all the patches, this section alternatively reports an overview of representative patches, which are nearer centroids for each cluster. LDA estimates a set of words that frequently appear in similar contexts among all the patches. In addition, each centroid for a cluster roughly reflects the average content of patches

Table 4.2: Cluster Examples
From left, ID of a cluster, number of patches, and three most probable topics of centroids for a cluster. Each topic is expressed by two most probable words of a topic (if highest probability is >0.9, expressed only by the word).

ID	Size	3 top topics for cluster
http	7021	(http, org), (fault, show), (id, cd)
thank	8921	(thank, cc), (manag, appli), (miss, add)
tx	5005	(tx, rx), (queue, blk), (packet, skb)
irq	5334	(irq), (interrupt, msi), (handler, c)
dma	5514	(dma, channel), (map), (id, cd)
x86	5005	(x86, iommu), (pci, slot), (max, min)
arm	5331	(arm, mach), (h, asm), (omap, omap2)
drm	5025	(drm, radeon), (auto, engin), (i915, pipe)
pci	5348	(pci, slot), (bu, driver), (cmd, pcie)
page	8078	(page, insert), (map), (scan, direct)
lock	5697	(lock, unlock), (lock, poll), (lock, protect)
null	6955	(null, derefer), (pointer, cast), (close, cap)

in the cluster. Thus, I expect words that represent the most probable topics for each cluster to also represent popular topics among all the patches for the Linux kernel.

Table 4.2 shows examples of clusters and their topics. For example, topic "*(null derefer)*" in `null` cluster in Table 4.2 implies that there were a significant number of contexts where "null" and "dereference" appeared in patches for Linux kernel. Like `null` cluster, this section mentions obtained clusters by corresponding cluster IDs in Table 4.2 in this disseration. Note that LDA estimates word co-occurrences with probabilistic; they do not always co-occur but tend to be in similar contexts.

Each cluster is characterized by words that represent topics for the centroid of a cluster. The biggest categories of clusters have topics about operating system semantics, including common features (`irq` (interrupt requests), `dma` (Direct

Table 4.3: Nearest fault description of a cluster centroid.

Cluster `arm`, commit 9cff337 3rd paragraph Topic: (arm, mach), (watchdog, nmi), (specif, code)
So far as I am aware this problem is ARM specific, because only ARM supports software change of the CPU (memory system) byte sex, however the partition table parsing is in generic MTD code. The patch below has been tested on NSLU2 (an IXP4XX based system) with a patch, 10-ixp4xx-copy-from.patch (submitted to Linux-arm-kernel - it's ARM specific) required to make the maps/ixp4xx.c driver work with an LE kernel.

memory access), and `page`), devices (`tx` (network transmissions), `drm` (direct rendering manager), `pci`, `x86`, and `arm`). Furthermore, there are general, well-known faults such as `lock` and `null` in Linux. Other clusters consist of words that often show up in development discussions such as `http` and `thank`.

Table 4.3 shows an example of the bug description and calculated topics in cluster `arm`. Words that represent topics characterized bug descriptions.

This study also investigated more than 20 patches for each cluster and confirmed that the centroid topics mostly represent the patches of each cluster. Since the concern of this dissertation is faults in operating systems, the study first filters out 30 uninteresting clusters before manual investigation. Then, some clusters are ignored if they obviously had topics unrelated to fixes, for example, `thanks`, `com`, and `comment` clusters. Other examples of ignored clusters are kernel-builds such as `build` and `select` clusters. `Powerpc` cluster were mostly relevant to builds, so this dissertation does not show the results.

In the Linux kernel, most of the topics were about device controls. Table 4.4 shows the clusters for controling devices. The table shows topics for 1) disks: ATA "*(ata libata)*", SCSI/Fiber Channel "*(scsi fc)*", and Block I/O "*(bio segment)*", 2) graphics: direct rendering manager/Radeon GPU "*(drm radeon)*" and other video "*(media video)*", 3) networks: transmission "*(tx rx)*", Ethernet "*(flow ethtool)*", and wireless networks/access points "*(ap beacon)*", and 4) Bus: PCI "*(pci slot)*" and USB "*(usb gadget)*". This result indicates Linux developers spent more efforts

Table 4.4: Clusters about device controls
From left, ID of a cluster, number of patches, and three most probable topics of centroids for a cluster. Each topic is expressed by two most probable words of a topic (if highest probability is >0.9, expressed only by the word). Values after topics are the probabilities of the topic occurrences in centroids (1.00 means 100% occurrence of the topic).

ID	size	most probable 2 topics (top 2 words)
block	5733	(block transact):0.088 (queue blk):0.031
bio	7029	(bio segment):0.056 (mark receiv):0.041
ata	5002	(ata libata):0.252 (id cd):0.025
scsi	5436	(scsi fc):0.153 (save sa):0.018
ap	5167	(ap beacon):0.169 (frame mac80211):0.028
tx	5005	(tx rx):0.137 (queue blk):0.017
flow	5909	(flow ethtool):0.197 (tx rx):0.013
drm	5025	(drm radeon):0.242 (auto engin):0.02
media	5062	(media video):0.146 (v4l2 sensor):0.023
usb	5135	(usb gadget):0.158 (cpufreq frequenc):0.02
pci	5348	(pci slot):0.116 (bu driver):0.011
x86	5005	(x86 iommu):0.155 (pci slot):0.008
arm	5331	(arm mach):0.116 (h asm):0.012
s390	5186	(s390 facil):0.132 (dirti nr):0.108
sh	5777	(sh migrat):0.23 (info displai):0.013
dvb	7018	(dvb v4l):0.214 (media video):0.059
soc	5790	(soc asoc):0.141 (detect pin):0.012
quirk	6493	(quirk laptop):0.161 (report hid):0.094

on commodity devices. On the other hand, there were some topics for specialized hardware like system-on-a-chip "*(soc asoc)*", digital video broadcasting/video for Linux "*(dvb v4l)*", and laptop-specific devices "*(quirk laptop)*". most of the SoC and laptop-specific devices were often sound devices and keyboards, respectively.

Table 4.5: Clusters about operating system features
From left, ID of a cluster, number of patches, and three most probable topics of centroids for a cluster. Each topic is expressed by two most probable words of a topic (if highest probability is >0.9, expressed only by the word). Values after topics are the probabilities of the topic occurrences in centroids (1.00 means 100% occurrence of the topic).

ID	size	most probable 2 topics (top 2 words)
net	5557	(net linu):0.126 (ocf2 truncat):0.11
nf	5804	(nf netfilt):0.178 (addr ipv6):0.017
socket	5498	(socket y):0.187 (addr ipv6):0.015
irq	5334	(irq):0.158 (interrupt msi):0.017
dma	5514	(dma channel):0.166 (map):0.011
pm	7843	(pm resum):0.077 (serial consol):0.074
memori	5296	(memori leak):0.152 (cpu hotplug):0.01
page	8078	(page insert):0.038 (map):0.018
crypto	5385	(crypto bss):0.256 (o drive):0.011
fs	5211	(fs uml):0.083 (declar static):0.082

Another category of device topics in Linux was CPU/platforms. There were topics "*(arm mach)*", "*(x86 iommu)*", "*(s390 facil)*", and "*(sh migrat)*" among the Linux patches. Recent trends towards virtualization were reflected on topic "*(x86 iommu)*", however, I did not observe virtualization-related topics in other clusters. An example document for `arm` cluster is shown in Table 4.3.

Table 4.5 shows that there are topics for operating system features such as page "*(page insert)*", interrupt "*(irq)*", DMA "*(dma channel)*", and power management "*(pm resum)*" in Linux. In addition, there are also topics for other software stacks. In particular, topics for network protocol stacks were significant as shown in topics for network filtering "*(nf netfilt)*", miscellaneous network-related materials "*(net linu)*", and sockets "*(socket y)*". Topic *(crypto bss)* indicates cryptography was also developed eagerly in the Linux kernel. Another software topic was about

Table 4.6: Clusters about general software
From left, ID of a cluster, number of patches, and three most probable topics of centroids for a cluster. Each topic is expressed by two most probable words of a topic (if highest probability is >0.9, expressed only by the word). Values after topics are the probabilities of the topic occurrences in centroids (1.00 means 100% occurrence of the topic).

ID	size	most probable 2 topics (top 2 words)
alloc	6318	(alloc):0.111 (memori leak):0.019
null	6955	(null derefer):0.071 (pointer cast):0.069
lock	5697	(lock unlock):0.088 (lock poll):0.055
bit	5133	(bit):0.105 (mask bit):0.05
limit	5314	(limit increas):0.066 (number calcul):0.049
debug	5745	(debug messag):0.128 (level format):0.018
timer	7763	(timer idl):0.109 (watchdog nmi):0.097

debugging/diagnosing like perf "*(perf counter)*", and kernel messages "*(debug messag)*". Unfortunately, there were no outstanding topics for file systems. "*(fs uml)*" represented not only file systems but also User-mode Linux. However, the cluster for transactions on block devices "*(block transact)*" gathered problems on journaling and block handling. An example patch for `irq` cluster is shown in Appendix A.2.

Finally, Linux kernel had topics for well-known programming materials (e.g., memory, lock) as shown in Table 4.6. For example, there are topics for NULL "*(null deref)*", lock "*(lock unlock)*", allocation "*(alloc)*", and timer "*(timer idl)*". Unfortunately, `memori` cluster did not contain a quite number of memory leaks, but it consists of problems on memory managements such as look-ups of OOM killers. In addition, there are arithmetic mistakes on data in `bit` and `limit` clusters. An example patch for `memori` cluster is shown in Appendix A.1.

```
//return 0 on success (negative on failures)
int request_irq(
  unsigned int irq,       // Interrupt line
  irq_handler_t handler,  // interrupt handler func.
  unsigned long irqflags, // Interrupt type flags
  const char *  devname,  // Device name
  void * dev_id);         // Device identity

void free_irq(
  unsigned int irq,       // Interrupt line
  void * dev_id);         // Device identity
```

Figure 4.3: Two checked API declarations and comments for each argument (declared in include/linux/interrupt.h)

4.4 Extracting Faults in Interrupt Handling

As a demonstration, developers can learn fault patterns from cluster `irq`. This section focuses on patches containing the topic "(free, descriptor)", which appears in 331 patches in this cluster. In the 331 patches, 160 are identified as device driver faults. Most fault patterns are identified as mistakes on the release of interrupt request handlers (IRQs) in device drivers. In the observation, request_irq(), request_threaded_irq() and free_irq() are frequently used APIs. To classify reports, the words representing the topics of each report are useful. For example, a report is classified as "irq leak on an error path during a device probing" because the report was represented by five topics of which the highly probable words are "(memori, leak)", "(irq)", "(probe, driver)", "(error)", and "(path)". Note that topics do not always reflect a fault directly, so this section performs manual analysis to learn the faults precisely.

4.4.1 API Semantics and the Programming Model

Before discussing the investigated faults, this section briefly describes API specification for ease of understanding the issue discussed. API specifications this section focuses on are in Figure 4.3. Request_irq() and free_irq() are in-kernel APIs for the registration of IRQ handler in Linux. They require various arguments, including an interrupt request number (irq), a flag of interrupt types, a function pointer for the interrupt handler corresponding to the irq, and an extra variable (dev_id). Free_irq() is an API for releasing a requested irq by specifying the irq and dev_id. Linux uses dev_id to validate requesting and releasing an irq shared among multiple drivers.

Also, it is necessary to consider the programming model of Linux device drivers so as to understand the problem. In particular, Linux device drivers often offer event-driven programming. The Linux kernel core dynamically invokes driver callbacks that the driver initialization routine registered for each external event such as physical device probe, removal, and power management.

Figure 4.4 shows typical API usages in device drivers to be checked. Drivers often store driver-specific states like IRQ numbers to given callback arguments (struct X *x in the example). The usage of the APIs is similar to that of other typical pairwise APIs (e.g., malloc/free, lock/unlock). However, there are some differences for checker implementation in practice. For example, when checking a shared IRQ, it is necessary to validate the consistency of two arguments unlike malloc/free, lock/unlock. Another difference is that drivers know the IRQ number before calling request_irq(). This means it is necessary to avoid accidental free_irq() on request-failed IRQs while failed malloc() returns NULL pointer that free() ignores. In practice, catching request_irq() failures is more error-prone than expected, especially when requesting multiple IRQs as shown in an example in Section 4.4.2.

4.4.2 Fault Patterns

Table 4.7 shows the observation results of faults that the clustering extracts. There are four fault types: `Argument`, `Leak`, `DoubleFree`, `Order`. Inconsistent arguments (`Argument`) is the major category of fault patterns. Missing free_irq()

```
int x_probe(struct X * x) {
    /* various resource initializations */
    ...
    if (request_irq(x->irq, x_isr, ..., x))
        goto err1;
    x->some_src = some_src_alloc();
    if (!x->some_src)
        goto err2;
    return 0;
err2:
// request_irq() must be revoked on failures
    free_irq(x->irq, x);
err1:
    /* release resources */
    return err;
}

void x_remove(struct X * x) {
    /* various release resources */
    ...
    free_irq(x->irq, x);
    ...
}
```

Figure 4.4: Typical API usages

is the second largest category in the observation (`Leak`), although `Argument` faults consequently cause the same effect as `Leak` faults. Similar to general faults, double-frees (`DoubleFree`) and order violations of releasing resources (`Order`) were observed. `Order` types had seven cases that were not for interrupt handler registrations. Table 4.7 shows all but `Order` type were the wrong usages of a paired API.

Figure 4.5 shows an `Argument` fault in the unified diff format of its fix. One of the typical mistakes of freeing IRQs occurs during the failure paths like that in

Table 4.7: Investigated faults
The table lists identified 160 faults in the cluster `irq`

fault type	Description	Num.
Argument	free_irq() with inconsistent dev_id	41
Argument	free_irq() with an invalid irq number	25
Leak	missing free_irq() at driver initialization errors	25
Leak	missing free_irq() at driver unloading	13
Leak	missing free_irq() before device is suspended	6
DoubleFree	double free_irq()	9
Order	releasing other src before free_irq()	7
Order	releasing pages with interrupt disabled	7
Order	freeing shraed irq with interrupt disabled	5
Other		22
Total		160

the example. Typical device drivers initialize their IRQ handlers as well as other resources. However, the resource initializations might fail. This means drivers have to revoke all the acquired resources as if the system did not load the driver in such cases. Before fixing the fault in Figure 4.5, the driver frees only an IRQ after failing to request an IRQ although she intended to free all the allocated IRQs. These kind of faults in failure paths are difficult to find by testing in defaultging environments. In this case, requesting IRQs rarely fails; users may encounter such rare cases when they load a particular device driver that (un)intentionally uses the same IRQ number.

Figure 4.6 is an example of a `Leak` fault. The driver requests an IRQ when it opens a serial port and frees the IRQ when it shutdowns the serial port. Developers can check the example fault by loading and unloading the driver with specific models of Samsung system-on-chips that make s3c24xx_serial_has_interrupt_mask(port) true. However, not all the maintainers have the specific models, and the models might be rare or old ones in the future.

drivers/misc/max8997-muic.c, Linux 3.4-rc3, commit 3241d56edda5

```
max8997_muic_probe(...) {
    for (i = 0; i < ARRAY_SIZE(muic_irqs); i++) {
        ret = request_threaded_irq(...);
        if (ret) {
            ...
-           for (i = i - 1; i >= 0; i--)
-               free_irq(muic_irq->irq, info);
            goto err_irq;
...
err_irq:
+   while (--i >= 0)
+       free_irq(pdata->irq_base
+           + muic_irqs[i].irq, info);
err_pdata:
    kfree(info);
```

Figure 4.5: Example of Leak fault in an error path

Also, there are too many device drivers that handle IRQs as described in the next section. Thus, such runtime testing by using physical devices is time-consuming and not cost-effective for checking a large number of drivers.

Most faults in Table 4.7 potentially cause serious consequences in systems although they are difficult to test by running systems. For example, no other device drivers can use an IRQ number until the system shutdowns as the consequences of `Leak` faults like in the example in Figure 4.6. In the case of `Argument` faults, developers may also unintentionally release IRQ handlers in other running device drivers. In other words, they do not release an IRQ handler in `Argument` faults like the example in Figure 4.5. `DoubleFree` faults cause either just redundant executions or missing free_irq() calls, depending on the developers' intention. The manifestation of `Order` faults depend on the timing of device interrupts, context switches, and concurrent executions. For example, an interrupt handler may ac-

drivers/tty/serial/samsung.c, Linux 3.9-rc3, commit b6ad29355560

```
s3c64xx_serial_startup(struct uart_port *port) {
    ...
    ret = request_irq(port->irq,..., ourport);
    ...
}
...
s3c24xx_serial_shutdown(struct uart_port *port) {
    if (s3c24xx_serial_has_interrupt_mask(port)) {
+       free_irq(port->irq, ourport);
        wr_regl(port, S3C64XX_UINTP, 0xf);
...
```

Figure 4.6: Example of Leak fault depending on user inputs

Table 4.8: Callbacks for registering and releasing IRQs

Callers of request_irq()		Callers of free_irq()	
struct name::member name	Num.	struct name::member name	Num.
platform_driver::probe	398	pci_driver::remove	253
pci_driver::probe	329	platform_driver::probe	240
i2c_driver::probe	204	pci_driver::probe	210
net_device_ops::ndo_open	121	platform_driver::remove	175
spi_driver::probe	72	i2c_driver::probe	145
platform_driver::remove	62	net_device_ops::ndo_stop	116
pci_driver::resume	42	i2c_driver::remove	108
work_struct::func	38	comedi_driver::detach	102
pcmcia_driver::probe	34	net_device_ops::ndo_open	67
comedi_driver::auto_attach	31	spi_driver::probe	56

cess invalid heap memory, and memory-mapped I/O (MMIO) if the driver releases the MMIO and memory earlier than IRQ.

4.4.3 Candidates of Fault Sites

The examples in Figures 4.5 and 4.6 suggest mistakes on IRQ releases happen on device drivers, i.e., loadable kernel modules in Linux kernel. An existing tool to detect intra-procedural resource-release omissions [86] shows the cases for initialization failures like in Figure 4.5. However, the example also implies there can be more difficult cases involving user operations like in Figure 4.6.

To confirm the validity of the implication, I analyzed at which event device drivers call request_irq() and free_irq() in Linux 4.1-rc1. The analysis first detects the root functions whose call graph includes at least one of the calls of two API and their family such as request_threaded_irq(). Then, it searches for an event callback in a device-specific struct (e.g., pci_driver). Finally, the two analysis results are joined so as to obtain event callbacks that call the two API families. This chapter describes the details of extracting event callbacks in Section 4.6.2.

Table 4.8 shows a part of the analysis results. The biggest users of the IRQ handler APIs were generic device drivers (function pointers in struct platform_driver) and PCI drivers (function pointers in struct pci_driver). As the example indicates, drivers call request_irq() at driver constructions such as device probes, opens, and resumes. Free_irq() occurs at driver destructions such as device removal and failure paths in device probes. The result shows that checkers should consider not only API pairwise confined within an event callback like in Figure 4.5, but also API pairwise crossing event callbacks like in Figure 4.6. Also, it shows there are many kinds of device drivers that request IRQs that potentially cause faults described in Section 4.4.2.

4.4.4 Summary and Discussion

The observation of IRQ faults in this section shows that there are many imbalances of request_irq() and free_irq() in the real world. This indicates that developing static checkers specific to validating the balance of the pairwise like [86] can effectively reduce debugging efforts during device driver developments. This is because the fault characteristics offer high coverages that dynamic testing cannot

achieve.

However, Table 4.8 and past examples show that it is necessary to track API pairwise crossing event callbacks. This dissertation does not focus on `Order` faults in this work because they are less frequent than wrong API pairwise, and such timing-dependent faults should be covered by fuzzing tests like [36].

4.5 Static Analysis of IRQ handling

This dissertation uses symbolic execution for an inter-procedural, path-sensitive static analysis to find IRQ faults on the basis of the observation in Section 4.4. One of the biggest advantages of symbolic execution is it can achieve high coverage from normal paths to exceptional, rarely executed paths even without running a target system [82]. Table 4.7 shows 25 faults appeared on rarely-executed paths such as error paths in driver initializations. Furthermore, there are many `Argument` faults on failure paths like the example in Figure 4.5. Symbolic execution is also effective to check such inconsistency of symbolic (or concrete, if possible) values for corresponding arguments.

4.5.1 Workflow

Figure 4.7 shows the abstract workflow to detect IRQ faults. First, the tool generates checked code from the original driver code and specifications of driver lifecycle (i.e., event callback execution-flows) given by users. Then, the execution engine runs on each translation unit (e.g., single .c file and included .h files) until it completes analyzing all the translation units. During the execution, the analyzer simply checks the state of each IRQ handler to validate the API pair. After all the analyses have finished, the fault-finding tool summarizes fault reports in HTML formats.

In this dissertation, driver complete binaries are not analyzed because pairwise APIs like IRQ handling mostly appeared in the same .c file. In other words, the tool ensures that device drivers confine their unit of system rules [33] within a

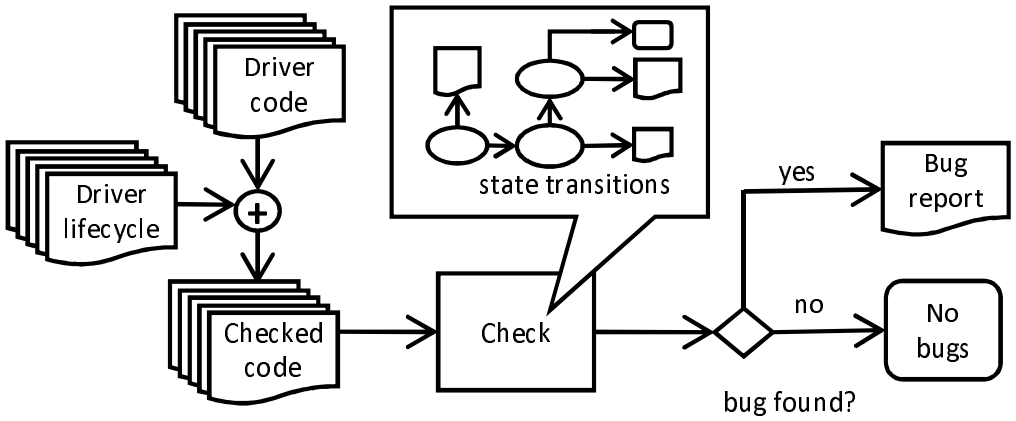

Figure 4.7: Analysis workflow

translation unit. For example, the tool alerts developers of a potential fault if a pair of request_irq() and free_irq() appears in different .c files.

4.5.2 IRQ State Tracking

Figure 4.8 shows the simplified version of an IRQ state transitions that the tool tracks and checks. At the beginning of analyzing a driver, the checker assumes the state is not tracked (*Untracked*). When the analyzed driver calls request_irq(), the analyzer stores the symbolic value for the arguments and moves the state to *Requesting*. Then, it bifurcates the state into *Requested* and *RequestFailed* for the succeeded and failed paths, respectively. *FreeRequestFailed* represents erroneous cases where drivers free non-existing IRQs specialized for request-failed (i.e., not-allocated) ones. By the bifurcation with symbolic executions, the checker can independently check two possible state transitions after request_irq() returns. Thus, after *RequestFailed*, the analyzer checks if free_irq() is called or not. Finally, it moves a *Requested* state to *Freed* when the driver calls free_irq() with the consistent symbolic values of the arguments. In current Linux, free_irq() never fails, and thus, there are no bifurcations after free_irq(). The report of a *Leak* fault appears when the driver does not call free_irq() for a *Requested* state at the end of the anal-

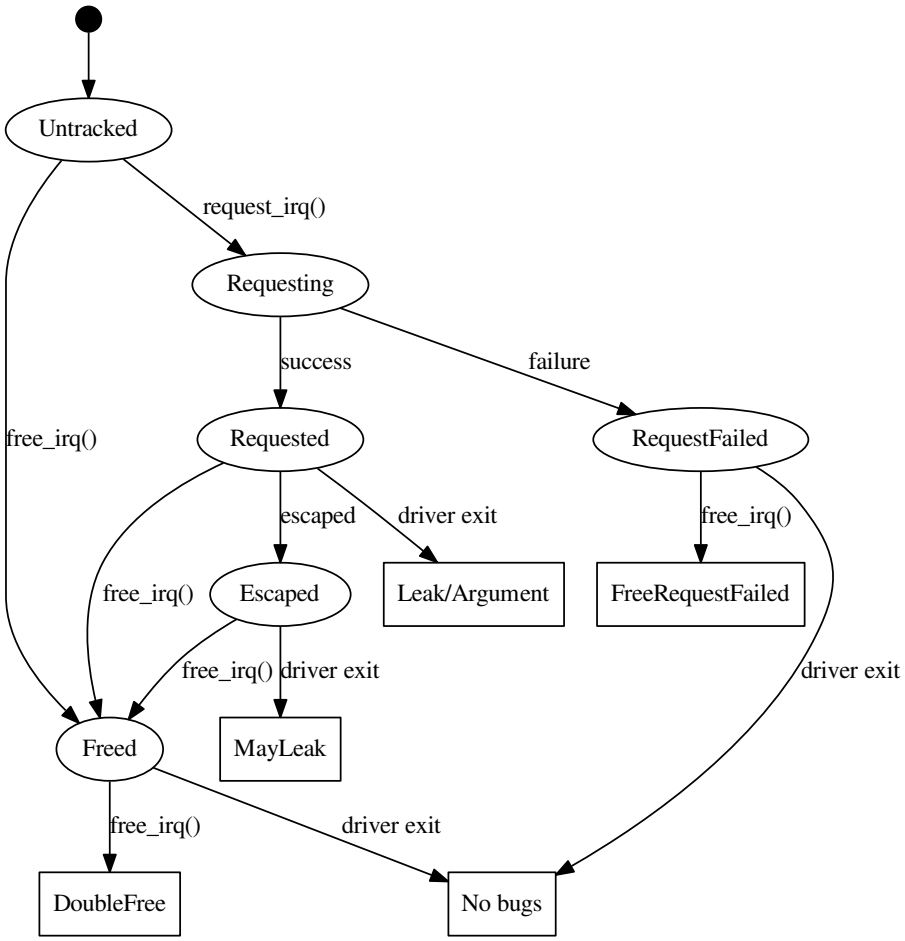

Figure 4.8: IRQ state transitions

Circles are the main states of IRQs, while rectangles are the consequence of state transitions

ysis. The symbolic execution engine keeps the value pairs even after freed IRQs so that *DoubleFree* faults can be reported if a driver frees an IRQ twice without requesting it again.

An analysis focusing on a translation unit potentially overlooks faults when an analyzed driver passes tracked states via pointers to external functions outside of a translation unit. For an example of IRQ numbers, drivers often store them inside a single struct variable instantiated for each driver (see the first argument of request_irq() in Figure 4.6). When the driver passes the pointer to the variable to an external function, the analyzer cannot detect the modification of tracked states by the function. As other checkers do in the Clang Static Analyzer, the issue can be mitigated by introducing an *Escaped* state. The *Escaped* state may confuse users by emitting false reports, but it enables users to prioritize inspecting reports with fewer false negatives. Thus, users can start their report inspections by using more doubtful reports such as *FreeRequestFailed*, *Double Free*, and *Leak/Argument* before less doubtful ones like *MayLeak*. Thus, the *Escaped* state is important for users to reduce manual efforts to find faults in large-scale, complex source code such as operating systems.

When the driver passes the pointer to the variable representing a *Requested* state to an external function, the analyzer turns the state to *Escaped*. The analyzer does not change states other than *Requested* because there are few external functions that call free_irq() i.e., *Double Free* and *FreeRequestFailed* rarely happen in external functions. The analyzer treats *Escaped* states like *Requested* states except for tracked values. For *Escaped* states, it tracks symbolic values of the address that stores the values that the analyzer tracked for the *Requested* state beforehand. When the analyzer detects the symbolic value of the address for arguments of free_irq(), it moves the *Escaped* state to *Freed*. Note that the analyzer reports all the analysis results that went through *Escaped* states (e.g., *MayLeak*) in order to avoid false negatives.

However, *Escaped* states cannot always be tracked. For example, it generates *Leak* reports when the address for the tracked state is potentially modified, although they might be false ones. This limitation affects the strategy of detecting Argument faults. In Figure 4.8, free_irq() to an *Untracked* state transits the state to a Freed state. The analyzer could report *Argument* faults when there was no consistent pair of arguments of free_irq() in stored arguments. In that case, how-

ever, escaped states cause false reports. Thus, the analyzer alternatively detects *Argument* faults as *Leak* faults.

4.5.3 Execution-flow Emulation

The previous section described how the analyzer manages and checks state transitions inside a static execution-flow. However, execution-flows of Linux device drivers are not always static because of the programming model of Linux device drivers.

To forge events at symbolic execution time, an emulating function is injected. The function simply calls registered callbacks for events in a typical order. It does not modify existing execution engines so that it increases the complexity of IRQ state tracking. Existing symbolic execution engines for C language can already emulate execution-flows that mainly appear in every function definition because of the nature of C language. Thus, it can emulate execution-flows at static analysis by adding a function that invokes callbacks in a typical event order.

To write the emulating function, it is necessary to identify the typical event order at runtime. This section describes it with an example of PCI device drivers in this section. A typical PCI driver execution-flow is shown in Figure 4.9.

Focusing on the exeuction-flow in Figure 4.9, the emulation code should first call probing callback in PCI device drivers immediately after the symbolic execution starts. Then, the symbolic execution engine bifurcates the checker execution into two because PCI device drivers in Linux sometimes fail to initialize their resources in probing functions. After probing devices, the driver might have to handle suspending or physical removal events. A suspending event can fail, so the execution engine bifurcates the checker execution as well as for probing events. On the other hand, removal and resuming events do not fail in Linux. Note that removal events can always happen physically except that Linux guarantees the event atomicity like resuming; device drivers have to handle even when users remove suspended devices or re-probe removed devices.

Figure 4.10 is the simplified version of injected code. The code is injected simply by appending a code fragment like in Figure 4.10 to existing .c files. Each

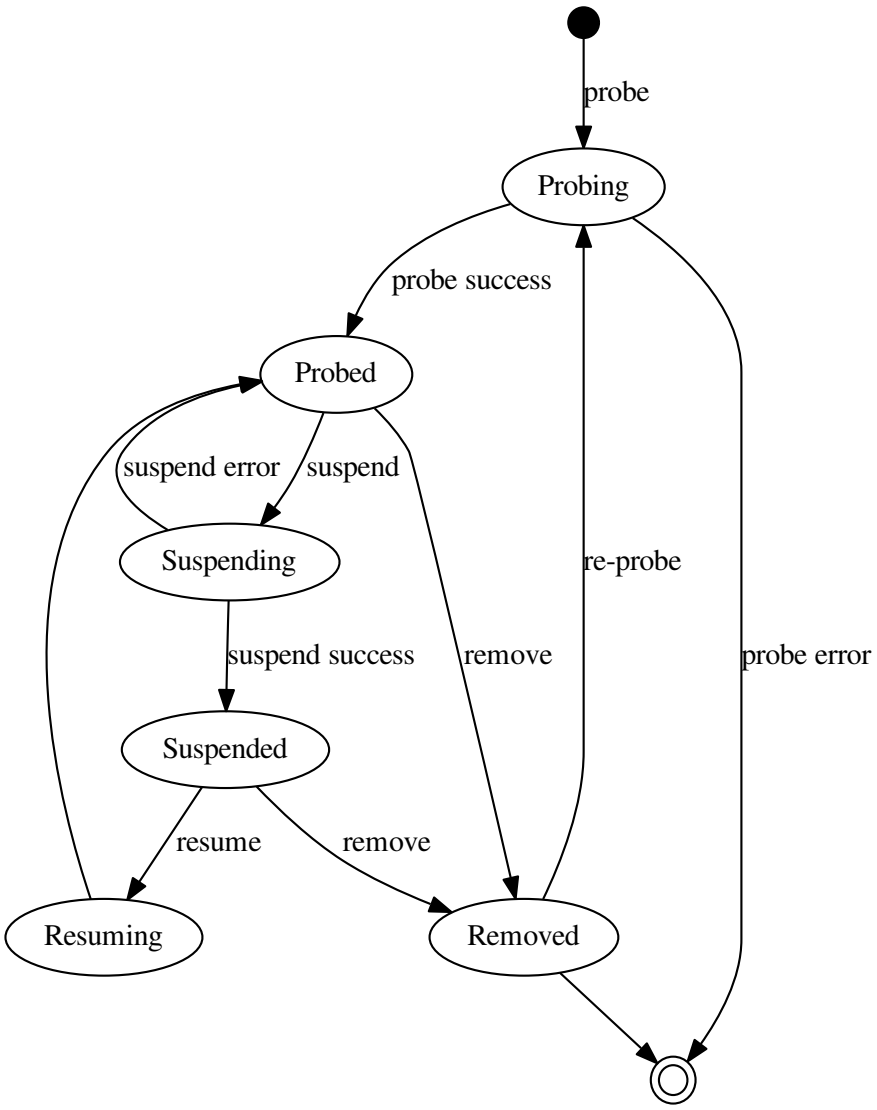

Figure 4.9: PCI driver execution-flow

```
#ifdef __clang_analyzer__
extern int random();
static void TestPCIDriver(struct pci_dev *pdev,
               const struct pci_device_id *id) {
        int loop = 0;
        enum PCI_STATE state;
reprobe:
        if (x_probe(pdev, id)) return;
        state = PCI_STATE_PROBED;

normal_operation:
        switch(random() % 4) {
        case PCI_EVENT_PM:
                x_power(pdev, &state);
                break;
        case PCI_EVENT_REMOVE:
                x_remove(pdev);
                state = PCI_STATE_REMOVED;
                if (random() % 2 && loop++ < 10) goto reprobe;
                break;
        default:
            /* do nothing */
        }
        if (random() % 2 && loop++ < 10
            && state == PCI_STATE_PROBED)
                    goto normal_operation;
        x_shutdown(pdev);
}
#endif
```

Figure 4.10: Example of injected code

x_probe, x_remove, etc. are calling pci_driver::probe(), remove(), etc. x_power is a function to emulate the state transition of suspends, hibernations, etc.

step of the driver invocation corresponds to driver callbacks (e.g., for a device probe, the injected code calls a struct pci_driver::probe()). The Linux documentation describes that the kernel core invokes PCI drivers by calling function pointers in struct pci_driver registered at the initialization of a device driver. The injector detects the defined callbacks by traversing abstract syntax tree to find initialization of function pointers in struct pci_driver as well as static analysis in Section 4.4.3. Templates of the specification are manually written as shown in Figure 4.10 and automatically replace x_* function with the functions obtained in the static analysis.

The code injection is to utilize symbolic execution properties. The injected code uses an external dummy function (random() in the figure) to bifurcate the execution into more than two. Also, symbolic execution engine stores a single generic driver state (pdev in the figure), and it can be shared among callback invocations by passing it as a function argument. In the Linux kernel, most of the drivers store dynamically allocated driver-specific states like IRQ numbers to the generic driver states. Thus, the injected function has parameters including driver state to create symbolic values for IRQ numbers in any invoked function. If the driver modifies the symbolic value for irq or dev_id after calling request_irq(), the tool can detect wrong usages of IRQ handling APIs.

Additionally, the code injection is to utilize rich C expressions to define driver execution-flows. For example, it uses a loop counter to conduct a bounded number of re-probing. It also uses a local variable to maintain the current state of the PCI driver. The extra requirements for learning domain-specific languages to define driver execution-flow are not necessary. Thus, the emulating function can be extended to other driver classes easily.

4.6 Implementation

The fault-finding tool is implemented as a plugin of Clang 3.7. The tool consists of two components: IRQ state tracker and code injector for emulating typical driver execution-flows in static analysis. The Clang Static Analyzer hooks compiler in-

vocations and runs the analysis with the code and the same compiler options. The analysis of the Linux kernel is performed with all options enabled (i.e., allyesconfig).

The IRQ state tracker described in Section 4.6.1 utilizes rich compiler-level information such as ASTs, call graphs, symbolic value information, and so on. The implementation consists of 1374 lines of C++11 and 264 lines of Python script. Clang provides developers the framework for customized static analysis including the symbolic execution engine.

4.6.1 IRQ State Tracker

During symbolic execution, the state tracker hooks the calls of request_irq() and free_irq() to track IRQ state transitions. However, to analyze only specified driver execution-flows, it ignores these two call expressions when the root of the traversed call sequence is not a specified entry function ("Test*" in the prototype).

At request_irq(), the state tracker makes new concrete value and symbolic value for both succeeded and failed return code of the API. Specifically, the succeeded value is zero (constant), and the failed value is a symbolic value for lower than zero. Then, the state tracker creates two execution contexts that add the new value constraints to the current symbolic execution state to emulate both succeeded and failure path of request_irq(). Request_irq() and free_irq() are external functions for checked drivers, and thus, it is necessary for the static analysis

When the execution hits an expression that calls APIs request_irq() or free_irq(), the state tracker extracts symbolic or concrete values for two arguments of the APIs: irq and dev_id. In the Linux kernel, paired values of the irq and dev_id represent IRQ identification. Thus, the analyzer uses symbolic or concrete values of them to identify if the released pair was already requested by the driver, for example. If the state tracker confirms an IRQ state remains *Requested* at the end of the entry function, it generates a Leak fault report.

Escaped states can appear on the expression of function calls with any pointers. After detecting the expression, the analyzer identifies type information of a pointed variable if the pointer passing potentially modifies values of irq and

drivers/net/ethernet/intel/e1000/e1000_main.c

```
static struct pci_driver e1000_driver = {
        .name     = e1000_driver_name,
        .id_table = e1000_pci_tbl,
        .probe    = e1000_probe,
        .remove   = e1000_remove,
#ifdef CONFIG_PM
        /* Power Management Hooks */
        .suspend  = e1000_suspend,
        .resume   = e1000_resume,
#endif
        .shutdown = e1000_shutdown,
        .err_handler = &e1000_err_handler
};
```

Figure 4.11: An example of callbacks

dev_id. If the escaped pointer can reach either irq or dev_id via struct member access or others, the analyzer marks the IRQ state *Escaped*. it assumes no buffer overruns that modify irq and dev_id because buffer overruns should be detected in other fault-finding tools. Other statements such as temporarily copying to global variables can cause *Escaped* states, but it does not handle such cases currently because such cases are few and can be recognized when developers manually check fault reports.

4.6.2 Code injector

Manual implementations in Figure 4.10 for each driver costs too much in terms of engineering. Thus, a code injector is implemented so that developers can automate the process of identifying the registered callbacks and generating injected functions.

Linux device drivers often register callbacks stored in constant variables like in

Figure 4.11. The injector recursively parses all the initialization statements to pick up declared function names passed as function pointer. Specifically, it looks up the left-hand side whose type is a function pointer. If so, it records the identifier of the left-hand side (with the struct type name), and the right-hand side. In the example, there are 'pci_driver', 'probe', and 'e1000_probe'. The callback interface allows developers to modify the callback functions, but the injector ignores such cases because drivers often do not change them dynamically.

After the callback identification, the injected code is generated from the template. The template is written carefully so that developers can emulate possible event orders as described in Section 4.5.3. For PCI drivers, the example code in Figure 4.10 is extended to track physical device errors that typical PCI protocol defines (err_handler in Figure 4.11). On the basis of the observation in Section 4.4, this dissertatoin focuses on 8 driver classes: generic (platform drivers in Linux), Peripheral Component Interconnect (PCI), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), network operations (open, close), control and measurement device interface (Comedi), and PCMCIA devices. Finally, 588 lines of C code are written for the template.

4.7 Experiments

In this experiment, emulation code is injected into 2287 drivers in the Linux 4.1-rc1 and finally checked 598 drivers that managed IRQ handlers. The tool generated 60 fault reports (i.e., *Leak/Arguments*, *Double Free*, and *FreeRequestFailed*), 177 *MayLeak* reports, and 294 *Escaped* reports. The 60 fault reports are more likely to contain faults because the analyzer completely tracked symbolic values related to IRQ API uses. *MayLeak* reports may contain *Leak* faults in which the analyzer detects no free_irq() calls with a requested IRQ, although it can track state transitions to *MayLeak* in Fig. 4.8. *Escaped* reports may contain faults that the analyzer cannot track (*MayLeak* reports are excluded), but are more likely to be false positives than other report types. This experiment runs on a single thread with Intel Xeon X5650 2.67GHz and 15 Gbytes RAM on HP ProLiant DL360

Table 4.9: Result
The table lists the overview of each fault the analyzer found. For precisely calculating the date of fault introductions (Since), the changes of file names are tracked. The beginning date of the Linux git repository. Thus, the lifetime of the fault in yenta_socket.c is longer than ten years.

Fixed file: drivers/power/88pm860x_charger.c Class: Generic, callback: platform_driver::remove() Fault type: DoubleFree, Path: Normal, Since Jul 27 2012
Fixed file: drivers/media/pci/ddbridge/ddbridge-core.c Class: PCI, callback: pci_driver::probe() Fault type: FreeRequestFailed, Path: Error, Since Jul 3 2011
Fixed file: drivers/pcmcia/yenta_socket.c Class: PCI, callback: pci_driver::probe() Fault type: Leak, Path: Error, Since Apr 16 2005*
Fixed file: drivers/power/wm831x_power.c Class: Generic: callback: platform_driver::probe() Fault type: FreeRequestFailed, Path: Error, Since Aug 10 2009
Fixed file: drivers/usb/gadget/udc/fotg210-udc.c Class: Generic, callback: platform_driver::probe() Fault type: FreeRequestFailed, Path: Error, Since May 30 2013
Fixed file: drivers/clocksource/sh_mtu2.c Class: Generic, callback: platform_driver::probe() Fault type: Leak, Path: Error, Since Apr 30 2009

G7. The static analysis required 13.2 hours for generating emulation code of driver lifecycles and 7.4 hours for checking state transitions.

The manual investigation started with the more suspicious of the 60 fault reports and then moved on to the less suspicious ones and found six cases of real faults within two weeks. When the analyzer detected suspicious code, a patch was written and sent to Linux maintainers in order to validate the results. Five out of

six patches were accepted and will be merged into the upstream version of Linux. At the time of writing, the patch for the bug for drivers/clocksource/sh_mtu2.c had not been accepted. This is because a developer responding to the report pointed out problems of the patch in code other than fixed IRQ handling.

Table 4.9 overviews the result of the checking. five out of six are on error paths at driver initializations. This is not suprising because developers can check normal paths for their drivers by reloading on their sites. However, one driver calls free_irq() twice at the normal path for a driver removal. Three FreeRequestFailed shows the effectiveness of path-sensitive analysis to validate the balances of two APIs.

Surprisingly, all the faults have existed since the driver was introduced into the Linux kernel. Thus, detected faults have survived a large number of code reviews, testing, and production runs for three to ten years. The faults potentially cause typical transient failures as discussed in Section 4.4.2.

Although the analyzer tries to prioritize emitted reports, there are a large number of false negatives. The 60 fault reports the analyzer detected contain one report for *DoubleFree* and three reports for *FreeRequestFailed* in Table 4.9 (=93.3% false positives). On the other hand, two *Leak* faults in Table 4.9 appear in 177 *MayLeak* reports (= 97.5% false positives). The 294 *Escaped* reports do not contain faults. Thus, there are at most 525 false positives out of 531 reports (= 98.9% false positives). Obviously, the false positive rate is very high. However, the analyzer reduces manual inspections from 2287 drivers to 531 reports.

Figure 4.12 shows the double free on the driver for a power charger. The fault can be found by simple intra-procedural analysis, but detecting it also requires loop extractions. The fix is to simply remove the redundant free_irq() before the loop.

Figure 4.13 shows a Leak fault in a Cardbus driver. Static analysis reported *MayLeak* on this fault because `socket->cb_irq` was potentially updated via a pointer `socket` in line 1256. At runtime, the failure can be manifested only when pcmcia_register_socket() fails and the device delivers an interrupt. Missing free_irq() lets the interrupt handler read from or write to resources freed by

drivers/power/88pm860x_charger.c

```
L739: pm860x_charger_remove(...)
L740: {
L741:   struct pm860x_charger_info *info = ...;
L742:   int i;
L743:
L744:   power_supply_unregister(info->usb);
-       free_irq(info->irq[0], info);
L746:   for (i = 0; i < info->irq_nums; i++)
L747:     free_irq(info->irq[i], info);
L748:   return 0;
L749: }
```

Figure 4.12: Code snippet for a *DoubleFree* on a power charger

the error handling of pcmcia_register_socket(). The fix is simply to add missing free_irq().

Interestingly, the example in Figure 4.13 also has four other resource-release omissions in the error paths for the driver probe. The maintainer found the problem and I fixed them as well as the IRQ leak. For example, a timer created by setup_timer(...) was not destroyed at the failure paths after the false condition of the first branch in Figure 4.13. This implies that the analyzer can even detect other kinds of faults that frequently co-occur at the same path of IRQ handling by validating IRQ handling.

Figure 4.14 shows a *FreeRequestFailed* fault. The code inappropriately unifies two error handling codes that are located after the goto label `fail1`. The checker tracked every paths and detected free_irq() after request_irq() fails, while it is difficult to reproduce the problem with dynamic testing. After request_irq() in line 1600, the symbolic execution engine in the Clang Static Analyzer bifurcates (or forks) the execution of checking state transitions. The bifurcated analyzer executions independently check state transitions after the transition to *Requested* or *RequestFailed*. The fix adds a branch condition before calling free_irq() because

drivers/pcmcia/yenta_socket.c

```
L1143: static int yenta_probe(...)
L1144: {

L1233:   if (... || request_irq(socket->cb_irq, ...)) {
L1235:     socket->cb_irq = 0;
L1236:     setup_timer(...);

L1243:   } else {
L1244:     socket->socket.features |= SS_CAP_CARDBUS;
L1245:   }

L1255:   dev_printk(KERN_INFO, &dev->dev,
L1256:       "..." ,cb_readl(socket, ...));

L1261:   ret = pcmcia_register_socket(...);
L1257:   if (ret == 0) {
L1259:     ret = device_create_file(...);
L1260:     if (ret == 0)
L1261:       goto out;
L1262:
L1263:       /* error path... */
L1264:       pcmcia_unregister_socket(&socket->socket);
L1265:   }
L1266:
+        if (socket->cb_irq)
+          free_irq(socket->cb_irq, socket);
L1267: unmap: /* pcmcia_register_socket failure */
L1268:   iounmap(socket->base);
L1275: out:
L1276:   return ret;
```

Figure 4.13: Code snippet for a *Leak* fault on a Cardbus driver

the driver should call free_irq() in the case where the driver in the *Requested* state

drivers/media/pci/ddbridge/ddbridge-core.c

```
L1563: wm831x_power_probe(...) {

L1600:   stat = request_irq(dev->pdev->irq, irq_handler,
L1601:     irq_flag, "DDBridge", (void *) dev);
L1602:   if (stat < 0)
L1603:     goto fail1;

L1610:   if (ddb_i2c_init(dev) < 0)
L1611:     goto fail1;

L1629: fail1:
L1630:   printk(KERN_ERR "fail1\n");
L1631:   if (dev->msi)
L1632:     pci_disable_msi(dev->pdev);
-        free_irq(dev->pdev->irq, dev);
+             if (stat == 0)
+                   free_irq(dev->pdev->irq, dev);
```

Figure 4.14: Code snippet for a *FreeRequestFailed* fault

encounters another failure in line 1610.

4.8 Summary

This chapter has studied faults in the Linux operating system. Natural language processing helped overview the past problems of Linux by grouping 370,403 patches into 66 clusters. The clusters represented the characteristics of patches in operating systems: operating system features, devices, and general software. As a case study, this chapter investigated the cluster for interrupt handling. The cluster extracted 160 patches for mishandling IRQs in device drivers.

The result indicated insights towards future directions of fault detection in operating systems. For example, most of the faults were not surprising ones;

existing tools for detecting and avoiding faults are expected to work effectively in real operating systems. On the other hand, the frequent observation of low-level semantics shows that testing and/or formal proof techniques with physical devices are desirable in some cases.

In addition to the study, this chapter presented an experience for checking fault patterns that are derived from the 160 patches. The checking succeeded in contributing five fault reductions in Linux device drivers. All the detected faults were serious but hard to find due to their non-deterministic and rarely executed properties. This chapter shows that utilizing software repositories is promising and should be further researched to enhance the future software quality.

However, there remains a gap between fault pattern recognitions and checker development to achieve practical fault avoidance. For example, all process for finding faults such as identifying faults and implementing static analysis are very labor-intensive. Also, even if they are automated in the future, the static checking spends too much time for checking a large number of fault patterns.

In addition, this study is just an initial work for understanding faults in real operating systems by using natural language processing. First, more manual investigations can be conducted. Second, there can be better parameters and algorithms of natural language processing. For example, clusters that have less than 5,000 patches and more than 10,000 can be investigated, although it is expected that inspections on such clusters result in more detailed and overviewed results compared to ours in this dissertation. The combination of supervised machine learning might speed up the process of studying common faults that reflect the real issues more exactly by filtering out more noises in the data set such as non-bug-fixes. As well as better parameter tuning, it can be effective to investigate narrow targets such as focusing on `fs/` directories and specific time periods. Finally, there are possibilities of studying other systems such as FreeBSD and Xen if the target system has a large number of documents such as issue trackers and changelogs.

Chapter 5

Error Propagation in the Linux Operating System

The objective of this chapter is to understand error propagation in Linux. Chapter 4 shows an overview of faults and results for six detected faults in Linux. However, Chapter 3 also shows that it remains necessary for advanced failure recovery to investigate operating system behaviors after software faults are activated.

This dissertation introduces the concept of error propagation scope. The propagation scope is *process-local* if the erroneous value is not propagated outside the process context that activated it. The scope is *kernel-global* if the erroneous value is propagated outside the process context that activated it.

This distinction between process-local and kernel-global errors is significant. If most errors are process-local, the kernel can recover from most errors simply by killing and revoking the resources of the faulty process. This implies that the Linux kernel can be partially rejuvenated without rebooting the entire operating system because the kernel does not need to verify every kernel state. If most errors are kernel-global, the recovery becomes hopeless because corrupted global data structures must be recovered to continue processing. In this case, a mechanism isolating propagated errors should be developed rather than recovery mechanisms.

5.1 Fault Injector

This chapter investigates error propagation scope in Linux under errors. To this end, an experiment was conducted where faults were injected into Linux code to see how it reacts to them.

5.1.1 Overview

The injector [72] emulates low- and high-level programming mistakes specific to operating system kernels. It rewrites the binary code of the running kernel to inject each type of fault. These injected faults approximate the assembly-level manifestation of real C-level programming errors. For example, the injector emulates missing initialization by deleting instructions that are responsible for variable initialization.

The injector disassembles the binary of a randomly selected function in the kernel text segment. Since the injected faults are context-dependent, it analyzes the disassembled code and searches for proper locations to which each type of fault can be injected. The details of the faults are described in Section 5.1.2. The injector runs in the kernel and provides a system call interface to specify the parameters of fault injection.

The injector is widely used to evaluate and validate recovery mechanisms in the operating system research community. For example, it has been used to evaluate the fault tolerance of the file system cache [72], recovery mechanisms for device drivers [91] [90], and quick reboot-based recoveries [30] [104].

5.1.2 Injected Faults

The injector emulates 15 types of faults. For ease of understanding, Table 5.1 lists examples of injected faults at the C-language level.

INIT fault: INIT fault creates a situation where the initialization of variables is not done. To create such a situation, the injector deletes instructions which initializes a variable with a constant value. More concretely, it deletes an instruction

Table 5.1: C-Language Level View of the Injected Software Faults.
This table shows examples of the injected faults at the C-language level.

Fault	Before	After
INIT	`int x = 0;`	`int x;`
IRQ	`arch_local_irq_restore()`	deleted.
OFF BY ONE	`while (x < 10)`	`while (x <= 10)`
BCOPY	`memcpy(p, p2, 256)`	`memcpy(p, p2, 512)`
SIZE	`kmalloc(256, ...)`	`kmalloc(128, ...)`
FREE	`kfree(p)`	deleted.
NULL	`if (p == NULL) return;`	deleted.
BRANCH	`if (cond) return;`	deleted.
DST&SRC	`x += 1;`	`x += 2;`
INVERSE	`if (cond) {...}`	`if (!cond) {...}`
PTR	`q = p->a;`	`q = p->b;`
VAR	`f(){char x[128];...`	`f(){char x[4096];...`
ALLOC	`p = kmalloc(...)`	`p = NULL`
LOOP	`while (x < 10) {...}`	`while (x < 20) {...}`
INTERFACE	`func(1, 2, 3);`	`func(1, 214, 3);`

that assigns an immediate value to the address lower than the stack pointer.

IRQ fault: When an IRQ fault is injected, the injector creates a situation where a kernel developer forgets to enable interrupts after disabling them. The injector removes `arch_local_irq_restore()` calls in Linux 2.6.38. When the call is removed, the interrupt mask is not restored, and thus, the disabled interrupts continue to be disabled. Note that the removed function should be updated for other versions of the Linux kernel. For example, in Linux 2.6.18, `local_irq_restore()` calls are removed.

OFF BY ONE fault: This fault imitates loop boundary condition errors. The injector changes conditions such as > to >=, < to <=, and so on. For example,

"jae" is changed into "ja".

BCOPY fault: Linux developers often mistake the use of mem* functions as well as application developers. The cases are emulated by BCOPY faults. The injector first searches a mem* function call at the random address of the kernel text. Then, it mutates an instruction that stores the constant value for the size of the copy to a register for the last argument. When the specified value for the last argument is not constant, the injection attempts to search another instruction for a mem* function call.

SIZE fault: buffer overruns happen when the size of heap allocations are insufficient. The fault injection also emulates the cases by searching kmalloc() call and mutating the first argument as well as it emulates BCOPY faults. kmalloc() is the most frequently used runtime API for dynamic memory allocations in device drivers.

FREE fault: This fault emulates a situation where the memory is not appropriately released. The injector removes the call to kfree(), which is responsible for releasing the unused heap memory. Since kfree() does not return any values, the injector simply deletes the call to kfree().

NULL fault: Forgetting NULL checks are observed in field studies of Linux faults [22] [76]. This fault injection mutates an instruction that conditionally jumps if the return value of a kmalloc() is zero.

BRANCH fault: This fault emulates an incorrect control flow by deleting a jump instruction involved in the conditional statement. By doing this, the injector emulates branch errors and error handling faults.

DST&SRC fault: This fault corrupts assignment statements. This creates a situation where the assignment is incorrect due to a programming error. To do this, the injector corrupts the value of the source or the destination by flipping the bits of the value.

INVERSE fault: The injector also reverses the predicates of conditional statements to inject incorrect control flows. For example, this fault changes "je" into "jne" to reverse the predicate.

PTR fault: This fault emulates pointer corruption by corrupting the address-

ing bytes of instructions. The injector either flips a bit within the addressing-form specifier byte (ModR/M) or the scale, index or base (SIB) byte following the instruction opcode.

VAR fault: The size of kernel stacks is limited in the Linux kernel. It causes stack overflows as existing work [22] observes. The cases are emulated by enlarging the size of a stack frame in a kernel stack. The injector searches two instructions that add and substitute the same immediate value to a stack pointer (`rsp`). Then, it increases the value of the immediate value for the two instructions. Adding and substituting `rsp` can also happen when the function calls with variable arguments such as `printk()`. However, the injector avoids such false emulations by searching it from the beginning and end of functions.

ALLOC fault: This fault makes `kmalloc()` return NULL to emulate the shortage of the heap memory. In x86_64, `kmalloc()` returns the address of the allocated memory through the `rax` register. Thus, `call kmalloc` is changed into `xor rax, rax` to inject the ALLOC fault.

LOOP fault: In x86, loops are often compiled into conditional jumps to negative offsets. Thus, the injector searches a conditional jump with a negative constant value and mutates the immediate value for the compare instruction that happen before the jump.

INTERFACE fault: This fault corrupts one of the arguments passed to a procedure. To create this situation, the injector deletes an instruction that copies a value at an address below the base pointer to registers or memory. For example, the injector can change the call `foo(a, b)` to `foo(X, b)`, where `X` is a corrupted value, by deleting the instruction that copies `a` to a register or memory.

5.2 Methodology

To investigate the scope of error propagation, the Linux kernel behavior is tracked when an injected fault is activated. To track how the Linux kernel reacts to the injected faults, this study takes the following steps manually:

(1) *Injecting a fault:* In the experiments, the text segment is modified to

inject faults as the target of this chapter is programming errors. The injected erroneous instructions may corrupt data in heap or stack. To trace the kernel execution after the fault is activated, a breakpoint is set at the instruction to which a fault is injected. When the breakpoint is hit the control is transferred to KDB, a built-in kernel debugger for Linux.

(2) *Running a workload:* This study uses six benchmarks that all stress the kernel. The six workloads are, 1) UnixBench on ext4, 2) UnixBench on fat, 3) UnixBench on USB, 4) Netperf, 5) Apaly, and 6) Restartd. UnixBench calls a lot of file- and process-related system calls and puts a heavy workload on current file systems. Netperf calls network-related system calls. Aplay invokes sound device drivers. As a benchmark, this study plays a wav file for 10 seconds. Restartd is a benchmark to restart all the system daemons. Since the daemons extensively issue system calls, the kernel code runs very frequently while the daemons are restarted.

(3) *Tracing error propagation:* After the fault is activated, the scope of error propagation is analyzed in the same way as taint analysis. If the kernel executes the injected fault and produces an erroneous value, the value is marked as an "error". When the value marked as an "error" is used to calculate another value, the calculated value is also marked as an "error". If the value marked as an "error" is used in the prediction of conditional branches, all the values updated in the taken clause are marked as an "error". If no value marked as an "error" is written to a heap, the error is concluded to be process-local. Otherwise, the error is concluded to be kernel-global. The kernel execution is tracked until kernel failures (e.g., kernel panic). If the workload finishes successfully, the failure is classified into "not manifested".

The previous study using fault-injection shows crash latencies are within 10 cycles in most cases [41]. However, this ignores, for instance, aging-related failures. The workload used in this experiment runs only within less than one minute. To cover these failures, this chapter shows the brief discussions of not-manifested errors in Section 5.4.

Note that error propagation is investigated at the assembly code level in the experiments, although this section describes the analysis of error propagation at

the source code level for readability. Error propagation can be analyzed more precisely if it is analyzed at the assembly level. For example, compilers generate optimized code that shares common expressions. Suppose that there are two expressions: x = a + b and y = (a + b) * c. If a fault is injected into the former a + b, it propagates to the latter.

5.3 Experiments

5.3.1 Experimental Setup

In this chapter, an experiment of fault injection is carried out on VMware Workstation 8 running on Windows 7. It runs Fedora 8 (Linux 2.6.38) in a guest virtual machine that consists of 1 CPU, 1 GB of memory and 20 GB hard disk drive. The host CPU is 2.53 GB Core2 Extreme CPU. The kernel configuration is default. Note that the failures encountered in this experiment are triggered by injected faults, not faults in the Linux kernel. The target system runs on a VMware workstation to reduce the time for rebooting the kernel after failures. The VMware workstation sometimes detects a critical error in the guest operating system and terminates the execution of it.

5.3.2 Error Propagation Scope

Figure 5.1 and 5.2 shows the overall results of the fault injection experiments. There are a total of 6,738 faults injected in the experiments and 13% of the injected faults are activated.

In the Linux kernel, there are two crash procedures: “kernel oops” and “kernel panic”. The kernel oops are called in 14% (124 out of 887) and panic() is called in 1.1% (10 out of 887). Linux kernel oops is invoked when the kernel detects an erroneous state inside itself. It kills an offending process and allows Linux to continue its operation under a compromised reliability. Kernel panics happen when the crash procedure fails. For example, Linux enters a panic state when it crashes during interrupt contexts because they cannot be safely terminated.

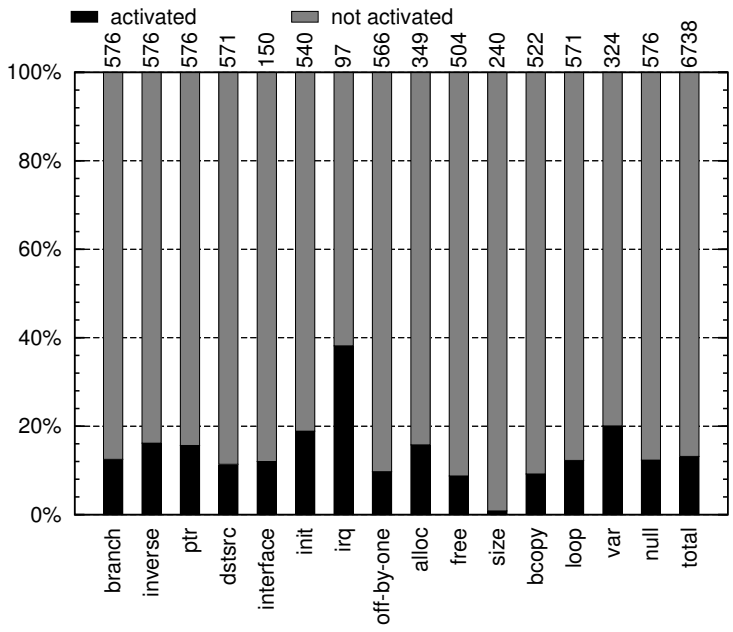

Figure 5.1: Activated/Not Activated Faults

This figure shows the relative frequency with which injected faults are activated or not. The number at the end of each bar represents the total number of injected faults.

However, Linux does not always detect failures. There were three types of failures that Linux did not detect. 9.9% of the manifested errors result in fail silence violations ("FSV"), hangs, and unexpected terminations by VMM ("TERM"). 75% of the faults are not manifested.

Figure 5.3 shows the result of error propagation scope for the 124 kernel oops and the 10 panic (because most panic() is called by the oops procedure). According to the experiments, 73% (98 out of 134) of the kernel oops are process-local, while 27% (36 out of 134) of them are kernel-global. This suggests that three quarters of the kernel oops can be recovered simply by revoking the faulty process. For BCOPY and SIZE faults, over 50% of errors are kernel-global because of buffer overruns.

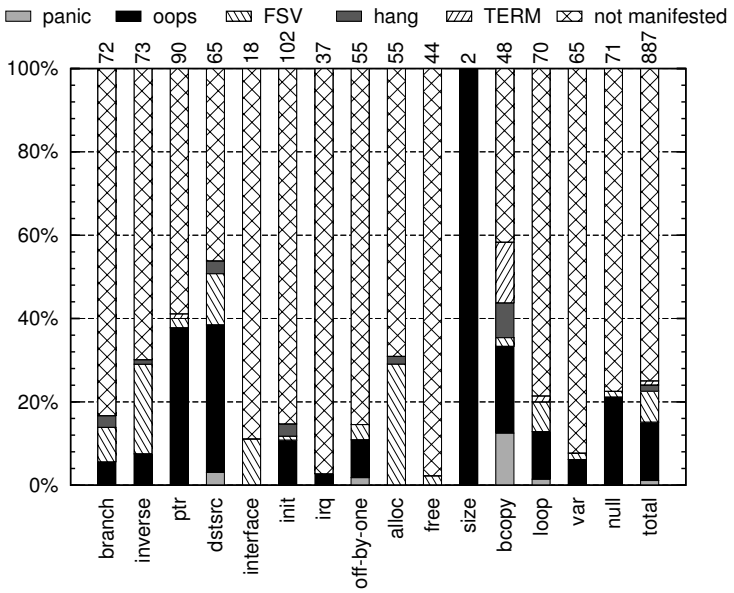

Figure 5.2: Observed Failures

This figure shows the relative frequency with which activated faults manifest different categories of failures. The number at the end of each bar represents the total number of activated faults.

This high rate of process-local errors is attributed to a defensive style of coding in Linux. For example, `BUG_ON` macro, which is similar to `assert()` in C, checks a given predicate and calls a kernel oops if the predicate is true. Some errors injected by the injector are caught by `BUG_ON` and their propagation is prevented.

5.3.3 Estimating Reliability after Kernel Oops

According to the results in the previous section, Linux is expected to be reliable with a probability of 73% after killing a failing process. This section investigates what happens if another workload runs after killing a faulty process. Just after killing a process, injected faults are removed to analyze the effect of the error propagation caused by it.

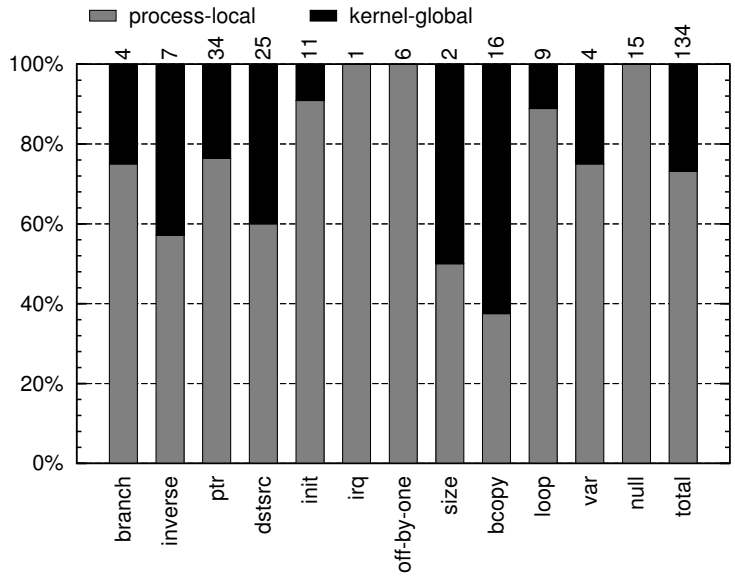

Figure 5.3: Error Propagation Scope

This figure shows the relative frequency with which propagated errors are process-local or kernel-global. The number at the end of each bar indicates the total number of investigated errors.

Figure 5.4 shows the summary of the kernel behavior after killing a faulty process. Note that killing a faulty process is the default crash procedure of the Linux kernel, which is called as the "kernel oops". Thus, the converge of the workloads run after the kernel oops are quite important for precisely estimating the reliability of the Linux kernel. To this end, an identical fault is injected again and again that caused the kernel oops in the previous experiment and run different workloads after the kernel oops.

No errors manifest in 68% of the process-local errors after the kernel oops. This probability is less than expected, where no errors manifest in almost all the cases. Even after the process-local errors, deadlock occurs in 29% (132 out of 463). This is because a faulty process is killed with the lock acquired. Although no global data structures are corrupted in process-local errors, the faulty process

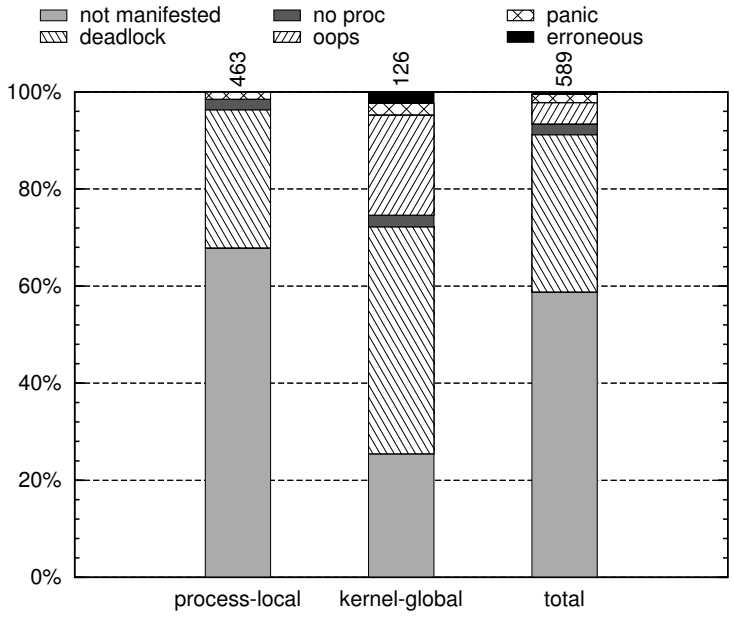

Figure 5.4: Kernel behavior after oops

This figure shows the relative frequency with which the kernel manifests different failure categories after oops recovery. The number at the end of each bar indicates the total number of investigated kernel behaviors. “erroneous” means cases where workloads fail due to internal errors of the Linux kernel without panic, oops, and other kinds of failures.

holds locks and killing it results in deadlocks after the kernel oops.

In kernel-global errors, no errors manifest in 25% after the kernel oops. This is because the workloads run after the kernel oops do not access the shared data corrupted by the faulty process. When the corrupted data is accessed after the kernel oops, deadlock occurs in most cases. In the experiments, deadlock occurs in 47%.

An error inside a critical section tends to result in a failure within the critical section because an error does not usually propagate a long way. Since the accesses to global data structures are controlled by synchronization primitives, the offending process is killed with the lock held and deadlocks are caused afterwards. This

behavior of the Linux kernel is preferable because it contributes to fail-stopness after the kernel oops. This result is interesting because no further data corruption occurs even after kernel-global errors in 72% ($= 25\% + 47\%$).

In summary, if Linux does not stop after the kernel oops, it runs reliably or stops its execution before trying to access corrupted data with a probability of 91% (not manifested and deadlock in Figure 5.4). While the kernel compromised by the process-local errors does not always succeed in continuing execution, kernel-global errors do not cause fatal failures in which the operation continues using inconsistent and corrupted data. In other words, the Linux kernel has a good fail-stopness property after the kernel oops.

Killing a faulty process sometimes leads to another problem. `No proc` in Figure 5.4 indicates cases where workloads running after the kernel oops cannot run as usual because the killed process is mandatory to continue the execution of the workloads. For example, UnixBench on USB cannot be started after kernel oops because a kernel daemon monitoring the plugs for USB devices is killed.

In process-local errors, panic() is called in 1.5% of the cases. It is observed when the kernel detects a buffer overrun in a kernel stack with a canary value, or the kernel finds that the faulty contexts are those for interrupts or the init process in the kernel oops procedure. The kernel determines to call panic() regardless of the state of its data structure, and therefore, panic() is observed even when errors are process-local.

In kernel-global errors, oops and panic() are called in 23% of the cases. In these cases, the errors that propagate to global data structures are simple, so access to them can be caught with the kernel oops. Unfortunately, there are three cases (labeled as `erroneous` in Figure 5.4) in which the Linux kernel continues its operation using inconsistent and corrupted data structures. However, this terrible situation happens only in 0.5% (3 out of 589 errors) of the cases in the experiments.

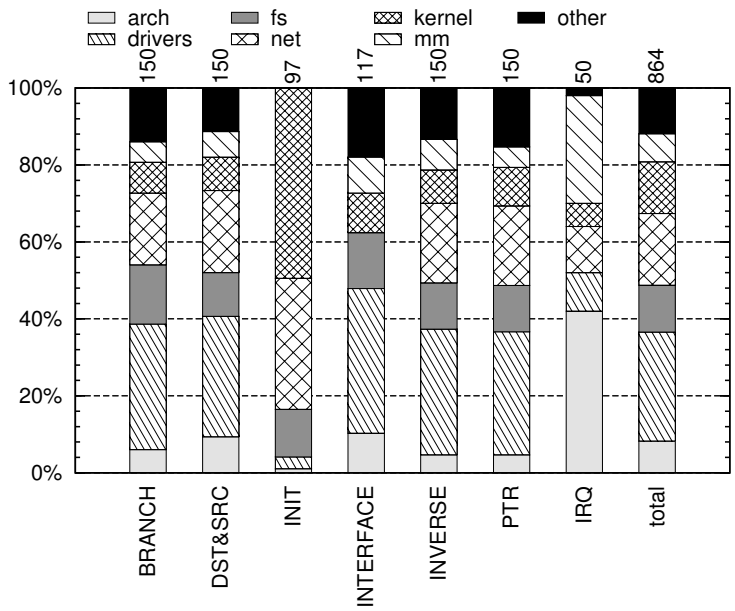

Figure 5.5: Fault Injection Sites

This figure shows the relative frequency on which directory in the kernel source code faults are injected. The number at the end of each bar represents the total number of injected faults.

5.4 Detailed Analysis of Error Propagation

Section 5.3 shows the overall results of error propagation scope before and after kernel crashes. This section reports in-depth observations of error propagation in Linux 2.6.18. The experiments in this section focus on seven faults in Table 5.1: INIT, IRQ, BRANCH, DST&SRC, INVERSE, PTR, and INTERFACE faults. Note that Section 5.3 discusses results with Linux 2.6.38 and 15 fault types, which is a different environment from the experiment in this section. However, the overall results of two different experiments did not significantly differ as shown later.

For the in-depth analysis, 864 faults are injected and 20% of them are activated in total. Figure 5.5 shows the ratio of fault injection sites in terms of the directory

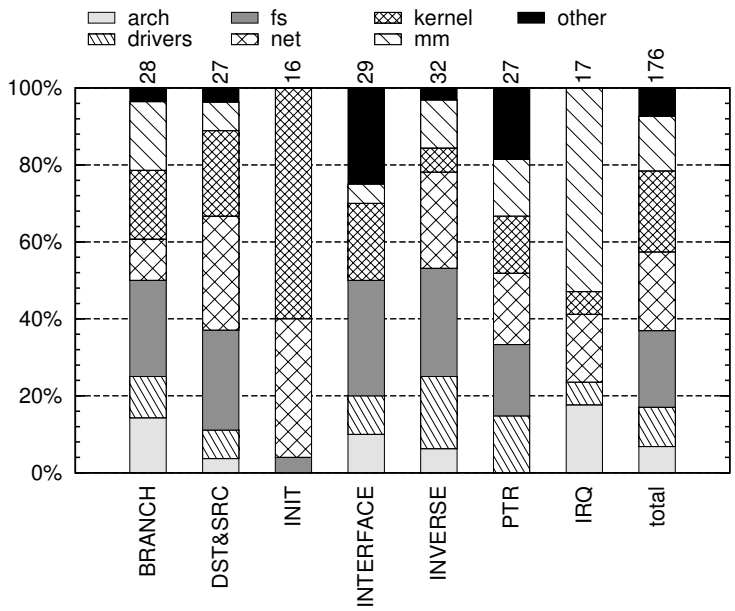

Figure 5.6: Fault Activation Sites

This figure shows the relative frequency on which directory in the kernel source code faults are activated. The number at the end of each bar represents the total number of activated faults.

name in the kernel source code. Figure 5.6 shows the ratio of fault activation sites in terms of the directory name. The fault injector selects injected location randomly from the kernel text segment, so the ratio depends on the size of each subsystem that is built in the kernel. INIT, INTERFACE, and IRQ faults cover relatively small number of directories because their target instructions (initializing with the base register and restoring the flags) are limited compared to other faults. These figures show the fault injection experiments cover all the common kernel subsystems. This result implies that injecting more errors will show the same categorization trend of errors.

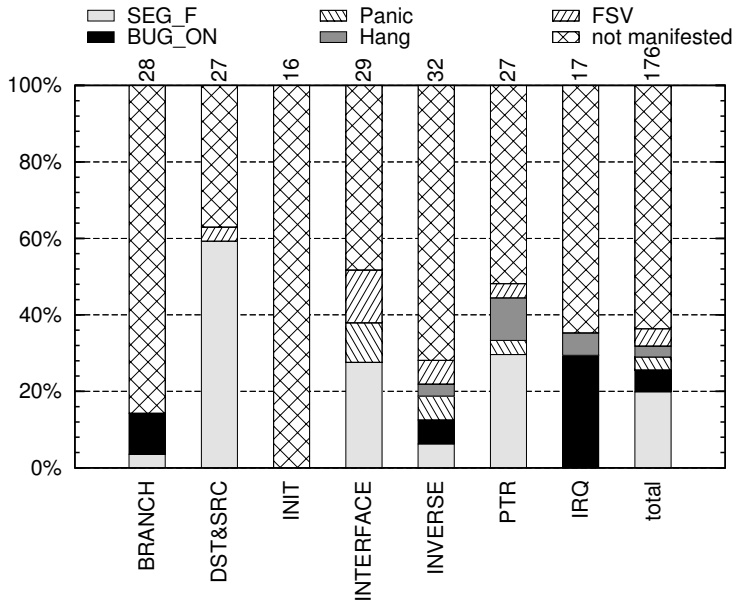

Figure 5.7: Observed Failures

This figure shows the relative frequency of not-manifested errors and the failure categories of manifested errors. The number at the end of each bar represents the total number of activated faults.

5.4.1 Failures

Figure 5.7 shows the failures which are observed after the fault activations. Segmentation failures ("SEG_F" in Figure 5.7) are caused in 20% of the fault activations. They occur when the kernel attempts to access illegal pages. Intentional kernel crashes caused by `BUG_ON` are observed in 6% ("`BUG_ON` in Figure 5.7). `BUG_ON` denotes a situation where Linux `BUG_ON` macro, similar to C assert, detects an erroneous state in the kernel. The other failures are panic, hangs and fail silence violations ("FSV" in Figure 5.7). 64% of the activated faults do not manifest themselves.

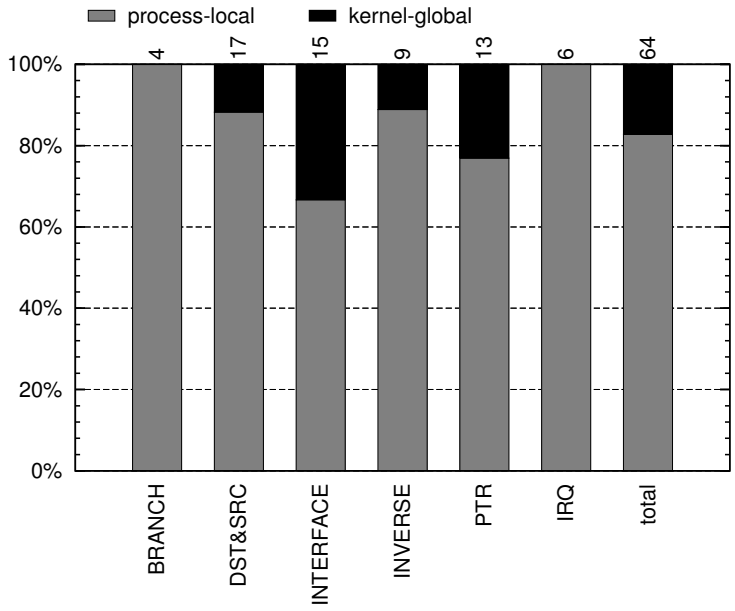

Figure 5.8: Error Propagation Scope

This figure shows the relative frequency of process-local or kernel-global errors. The number at the end of each bar represents the total number of investigated errors.

5.4.2 Error Propagation Scope

Figure 5.8 and 5.9 summarize the results of the scope analysis. 84% of the manifested errors are process-local, while 16% of them are kernel-global. BRANCH and IRQ faults are not propagated outside a faulty context, while INTERFACE faults are the highest rate of kernel-global errors. IRQ faults and their error propagation are described in Section 5.4.2 (b). INTERFACE faults tend to corrupt the linked lists for kernel descriptors. The typical case is shown in Section 5.4.2. The manifested BRANCH faults tend to be injected to a branch instruction for a NULL check. Checked pointers are usually used in the clauses. Thus, a crash tends to occur soon after the fault activation. The short crash latency leads to the errors which are confined within a faulty context.

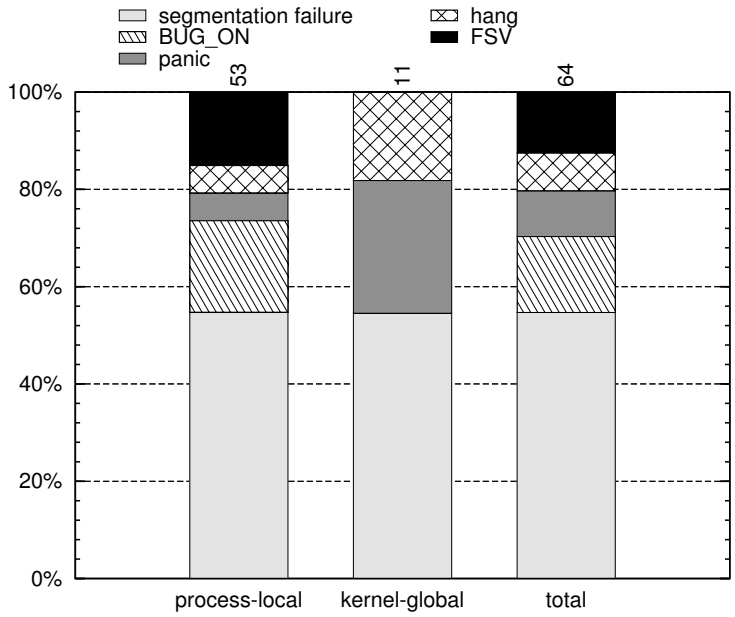

Figure 5.9: Failure Type by Scope

This figure shows the relative frequency with which the kernel causes different failure categories after fault activations. The number at the end of each bar represents the total number of investigated errors.

Figure 5.9 summarizes observed failures in terms of their error propagation scope. These segmentation failures occur in both propagation scopes with the highest probability out of all the observed failures (56% of all the manifested errors). All of the fail silence violations and `BUG_ON` are caused only by process-local errors. This result implies that `BUG_ON` effectively prevents global propagation in the kernel as described in Section 5.4.2.

This experimental environment uses ext3 file system, which is the default configuration of Fedora 8. Real bugs inside the kernel might destroy the file system structure. However, none of the experimental results showed such cases.

Table 5.2: Segmentation Failure

Fault	INVERSE FAULT
Memory Address	`sysfs_new_inode+0x5c`
Code Location	`fs/sysfs/inode.c, line:134`
Original Instruction	`je sysfs_new_inode+0x97`
Modified Instruction	`jne sysfs_new_inode+0x97`
Original Code	`if (sd->s_iattr) {`
Modified Code	`if (!sd->s_iattr) {`

Table 5.3: BUG_ON

Fault	IRQ FAULT
Memory Address	`kfree+0x5f`
Code Location	`mm/slab.c line: 3463`
Original Instruction	`push %esi popf`
Modified nstruction	`nop nop`
Original Code	`local_irq_restore(flags);`
Modified Code	deleted

Process-local errors

Table 5.2, 5.3 5.4, 5.5, and 5.6 show typical examples of each failure type caused by process-local errors. Each table lists an injected fault type, a memory address where the fault is injected, the location at the source code level, and the instructions and C-code before/after the fault injection.

(a) *Segmentation Failure:* As shown in Figure 5.9, 56% of the process-local errors lead to segmentation failures. Table 5.2 shows the detail of a typical case that leads to a segmentation failure. In this case, a null pointer is passed to a function that expects the passed pointer not to be null. This fault is injected by INVERSE FAULT. More concretely, the code

```
if (sd->s_iattr) {
   set_inode_attr(inode, sd->s_iattr);
   ...
```

Table 5.4: Panic

Fault	INTERFACE FAULT
Memory Address	`neigh_update+0x1ed`
Code Location	`net/core/neighbour.c line:894-895`
Original Instruction	`mov 0xc(%ebp), %eax`
Modified Instruction	`nop nop nop`
Original Code	`void (*update)(...) = neigh->dev-> header_cache_update;`
Modified Code	`void (*update)(...) = (struct netdevice *)(0x6)-> header_cache_update;`

Table 5.5: Fail silence violation

Fault	SRC&DST FAULT
Memory Address	`sock_alloc_fd+0xb`
Code Location	`net/socket.c, line:380`
Original Instruction	`mov %eax, %ebx`
Modified Instruction	`mov %esp, %ebx`
Original Code	`fd = get_unused_fd();`
Modified Code	`get_unused_fd();`

is modified to

```
if (!sd->s_iattr) { // FAULT injected here
   set_inode_attr(inode, sd->s_iattr);
   ...
```

In the original code, `set_inode_attr` is called only when `sd->s_iattr` is not NULL. However, `set_inode_attr` is called when `sd->s_iattr` is NULL in the modified code. As a result, parameter `iattr` in `set_inode_attr` becomes NULL as shown below. The dereference of `iattr` causes a segmenta-

Table 5.6: Hang

Fault	IRQ FAULT
Memory Address	`do_softirq+0x48`
Code Location	`kernel/softirq.c, line:215`
Original Instruction	`push %esi popf`
Modified Instruction	`nop nop`
Original Code	`local_irq_restore(flags);`
Modified Code	deleted

tion fault.

```
void set_inode_attr(inode, iattr)
{
    // Failure
    inode->i_mode = iattr->ia_mode; // iattr is NULL
```

In this case, a null pointer is passed across function calls but no global data structures are updated with the incorrect null pointer. Thus, the scope of error propagation is process-local.

(b) *BUG_ON:* As shown in Figure 5.9, 19% of the process-local errors lead to `BUG_ON`. An example of this failure (Table 5.3) is caused by IRQ FAULT, which removes the call to `local_irq_restore` to forget to enable disabled interrupts. After this fault is activated, the kernel continues to run with the interrupts disabled. Meanwhile, `lookup_bh_lru(bdev, block, size)` is invoked. This function eventually calls `check_irqs_on`, which executes `BUG_ON(irq_disabled())`. Since the interrupts are disabled here (if the fault is not injected, the interrupts are enabled here), `BUG_ON` macro successfully detects this incorrect status of interrupts.

This experimental result suggests that `BUG_ON` macro is effective to prevent global error propagation. If `BUG_ON` is not used to check the status of interrupts, blocking functions are called with the interrupts disabled and thus, the deadlock or other serious situations would result. In the current versions of Linux, `BUG_ON` macro is inserted manually according to the developers' experiences and intuitions. Thus, more systematic methods are required in order to help the developers

insert BUG_ON macros correctly.

(c) *Panic:* As shown in Figure 5.9, 6% of the process-local errors cause kernel panic. Table 5.4 shows a typical example of panic. In this case, a fault is injected into an interrupt handler. More concretely, an argument to function neigh_update is corrupted and thus the address of neigh->dev, which is calculated from the corrupted argument, becomes an incorrect value. As a result, the first access to neigh->dev causes a segmentation failure. Since this code is executed in an interrupt handler, the kernel invokes panic instead of causing a segmentation failure. Interrupt contexts temporarily use a kernel stack of the current process's kernel context in the Linux kernel. There were no structural differences between interrupt and processes' contexts. Therefore, this study regards the context as a process's context and the error confined in it is process-local.

(d) *Fail silence violation:* 15% of the process-local errors lead to fail silence violation as shown in Figure 5.9. In the experiments, Fail silence violations often derive from kernel error detections. Despite their correctness, the kernel starts to handle the detected errors by the usual error processing manner. Furthermore, such error processing tends to simply abandon the current processing and return a corresponding erroneous value (e.g., EINVAL). Therefore, global data structures are merely updated before fail silence violations occur. In the experiments, there were no kernel-global errors that lead to fail silence violations.

The following is a typical example of fail silence violation. In this example, the injected fault (Table 5.5) generates a situation in which there is no unused network sockets. Thus, the Linux kernel considers no network sockets can be created. The following is simplified code for explanation. The original code

```
int sock_alloc_fd(...) {
   int fd;
   fd = get_unused_fd(); // Fault injected here
   ...
   return fd;
}
```

is modified to

```
int sock_alloc_fd(...) {
   int fd;
   get_unused_fd(); // "fd =" is removed
```

```
    ...
    return fd;
}
```

In the modified code, `fd` is not initialized. In this experiment, uninitialized `fd` happens to be negative. As a result, `sock_alloc_fd` returns a negative value to its caller. The caller is:

```
// sock_alloc_fd is called here
// retval becomes negative
retval = sock_alloc_fd(sock);
// Linux considers no socket
// can be allocated
if (retval < 0)
   goto out_release;
...
out_release:
// socket is released and
// a negative value is returned
sock_release(sock);
return retval;
```

In the above code, the Linux kernel considers there is no room to create a new socket because `sock_alloc_fd` returns a negative value. As a result, a process cannot create a new socket even though there is enough room to create new sockets.

(e) *Hang:* As shown in Figure 5.9, 5% of the cases were when the Linux kernel hangs up. A typical example is shown in Table 5.6. In this example, IRQ FAULT is injected into `do_softirq` which schedules pending software interrupts. When `do_softirq` returns, the kernel hangs immediately without dumping the stack trace. Thus, the kernel behavior cannot be traced using KDB. Since it was unclear which function was executed after `do_softirq` returns, further information could not be obtained in this case.

Kernel-global errors

16% of the errors are kernel-global as shown in Figure 5.8, while all the other errors are process-local. Some of the process-local errors propagate across multiple function calls but the propagations are limited to function arguments, re-

Table 5.7: A Kernel-global error

Fault	INTERFACE FAULT
Memory Address	`rb_erase+0x1e9`
Function	`lib/rbtree.c, line:178`
Original Instruction	`mov 0x0(%ebp),%ebx`
Modified Instruction	`nop nop nop`
Original Code	`node = root->rb_node;`
Modified Code	`node = parent->rb_right;`

turn values, and local variables. This is probably because global data structures, shared among multiple processes, are used to store stable, consistent states rather than transient, temporary states. Experienced programmers like Linux developers write defensive code that checks data integrity and/or confirms the assumptions on function arguments. A data is checked again and again before it is written to global data structures.

Table 5.7 shows the detail of a representative kernel-global error. In this case, a fault is injected into a function that manages red-black trees, a type of self-balancing binary search tree, used for storing sortable key-value pairs. More specifically, INTERFACE FAULT is injected into the call to `__rb_erase_color`. Function `__rb_erase_color` takes three arguments: `node`, `parent`, and `root` whose types are all `struct rb_node*`. By the INTERFACE FAULT, argument `node` that should be `root->rb_node` is modified to `parent->rb_right`. As you can imagine from the arguments, `__rb_erase_color` manipulates tree structures in the heap. The incorrect argument leads to the corruption of the global tree structures. When the kernel traverses a broken red-black tree, it crashes due to segmentation fault. Since global data structures are corrupted by injected faults, the scope of this error is kernel-global.

There is one important thing to be noted. The fault shown in Table 5.7 corrupts global data structures. However, the erroneous values are never propagated to other processes than the faulty one. Other processes can continue to run reliably

because the error can be isolated. This is because the faulty process can hold a lock (more precisely, semaphore) for exclusive access to global data structures. If a faulty context does not release the lock, other processes cannot access the broken data structures; the corrupted data is never propagated to other processes. This example shows a deadlock leads to fail-stopping behavior although deadlock should be avoided. Deadlock prevents contexts from reading erroneous values, which may cause some incorrect kernel behavior like file system corruption. However, further research effort is required to apply this property of synchronization primitives to error recovery in practice.

5.4.3 Not-Manifested Errors

To understand Linux behaviors under errors, it is critically important to analyze the reason why activated faults do not manifest themselves. As pointed out in many literatures, activated faults do not always manifest themselves. In this experiment, These "not-manifested" errors are observed in 64% of the fault activations. If an error is corrected during the execution, the analysis aids in proposing defensive coding styles effective for kernels.

Table 5.8 shows the summary of the errors not manifested in this experiments. In this table, these errors are classified into 8 cases, based on the reason why they do not manifest themselves.

Corrected: "Corrected" indicates a situation in which an erroneous state is corrected by the Linux kernel. A typical example of this error is as follows. As shown in Table 5.9, a fault is injected to remove the initialization of `oldpolicy`. In the original code, `oldpolicy` is initialized to `-1`. This error is corrected as follows.

```
int oldpolicy; // should be initialized to -1
...
if (unlikely(oldpolicy != -1 ....)) {
   policy = oldpolicy = -1; // error corrected
```

Not affecting: "Not affecting" indicates a situation where an erroneous state is not used by the kernel. For example, a local variable is corrupted but not used at all until the end of the function after the injection, as described in Table 5.10.

Table 5.8: Summary of Not-Manifested Errors.
The table shows the number of errors for each reason that activated errors do not manifest themselves. An error is regarded as not-manifested when one of these situations is observed during the tracing of error propagation. The untraceable cases are discussed in detail.

Reason	# of errors
Corrected	8
Not affecting	10
Error processing omitted	18
Incorrect warning	4
Almost correct operation	15
Aging	6
Lucky	40
Untraceable	11
Total	112

Table 5.9: Corrected

Fault	INIT FAULT
Memory Address	`sched_setscheduler+0x44`
Code Location	`kernel/shed.c, line:4087`
Original Instruction	`movl $0xffffffff, 0xffffffec(%ebp)`
Modified Instruction	`nop nop ... nop`
Original Code	`int oldpolicy = -1;`
Modified Code	`int oldpolicy;`

In this example, local variable `all_pinned`, which is not initialized, is not used in this experiments until the function returns.

Error processing omitted: "Error processing omitted" indicates a situation where the code for error processing is omitted. This error does not manifest itself

Table 5.10: Not affecting

Fault	INIT FAULT
Memory Address	`rebalance_tick+0xda`
Code Location	`kernel/sched.c line: 2530`
Original Instruction	`movl $0x0, 0xfffffff0(%ebp)`
Modified Instruction	`nop nop ... nop`
Original Code	`int all_pinned = 0;`
Modified Code	`int all_pinned;`

Table 5.11: Error processing omitted

Fault	BRANCH FAULT
Memory Address	`follow_page+0xd8`
Code Location	`mm/memory.c, line:935`
Original Instruction	`je follow_page+0x1aa`
Modified Instruction	`nop nop ... nop`
Original Code	`if (!ptep) goto out;`
Modified Code	deleted

during the experiments unless the omitted error processing becomes necessary. The detail of a typical example of this case is shown in Table 5.11.

Incorrect warning: "Incorrect warning" indicates a situation where warning messages are displayed even though those messages should not be displayed. This is caused by the omission of conditional jumps that judge if warning messages should be displayed. The detail is shown in Table 5.12.

Almost correct operation: "Almost correction operation" indicates a situation where the kernel behavior is slightly changed from the expected one but the kernel continues to run as normal. Most of these errors are related to scheduling parameters that affect the scheduling behavior of the kernel. In the example shown in Table 5.13, the code for initializing local variable `run_time` is removed by fault injection. Since `run_time` is used to calculate the sleeping time of pro-

Table 5.12: Incorrect warning

Fault	BRANCH FAULT
Memory Address	`net_tx_action+0x37`
Code Location	`kernel/sched.c, line:2845`
Original Instruction	`je net_tx_action+0x55`
Modified Instruction	`nop nop`
Original Code	`if (unlikely(!(x))) {`
Modified Code	deleted

Table 5.13: Almost correction operation

Fault	INIT FAULT
Memory Address	`schedule+0xd2`
Code Location	`kernel/sched.c, line:3341`
Original Instruction	`movl 0x3b9aca99, 0xffffffc4(%ebp)`
Modified Instruction	`nop nop ... nop`
Original Code	`run_time = NS_MAX_SLEEP_AVG;`
Modified Code	deleted

cesses, it changes the scheduling behavior if set improperly. As shown below, even if `run_time` becomes erroneously large, the kernel code corrects the value. As a result, the kernel continues to run almost normally.

```
...
// Following statement removed
run_time = NS_MAX_SLEEP_AVE;
...
// prev->sleep_avg becomes incorrect here
prev->sleep_avg -= run_time;

// prev->sleep_avg corrected if necessary
if ((long)prev->sleep_avg <= 0)
   prev->sleep_avg = 0;
```

Table 5.14: Aging

Fault	INVERSE FAULT
Memory Address	`mousedev_release+0x37`
Code Location	`drivers/input/mousedev.c line:391`
Original Instruction	`jne mousedev_release+0x9a`
Modified Instruction	`je mousedev_release+0x9a`
Original Code	`if(!--list->mousedev->open){`
Modified Code	`if(--list->mousedev->open){`

Table 5.15: Lucky

Fault	PTR FAULT
Memory Address	`tty_register_driver+0x6`
Code Location	`drivers/char/tty_io.c line:3733`
Original Instruction	`mov 0x6c(%esi),%eax`
Modified Instruction	`mov 0x6d(%esi),%eax`
Original Code	`if(!driver->major){`
Modified Code	`if(!*(&driver->major+0x1)){`

Aging: "Aging" indicates a situation where resource leakage occurs. Software aging is a serious problem but the aging errors seem not to manifest themselves during the short duration of fault injection experiments. An example of aging is shown in Table 5.14. Before the fault injection, a reference counter is checked and the resource for a mouse device is released in this clause. Although this environment does not use mouse devices, the unreleased memory might pressure the kernel memory.

Lucky: "Lucky" indicates a situation where an error is activated but happens to cause nothing wrong. For example, INIT FAULT removes code for initializing a local variable to zero, whose value happens to be zero. Another example (shown

Table 5.16: Untraceable

Fault	INTERFACE FAULT
Memory Address	`rtnetlink_fill_ifinfo+0x2ec`
Code Location	`net/core/rtnetlink.c line:273`
Original Instruction	`mov 0x68(%ebp), %eax`
Modified Instruction	`nop nop nop`
Original Code	`u32 mtu = dev->mtu;`
Modified Code	`u32 mtu = dev->broadcast;`

in Table 5.15) is from `tty_register_driver`, which is used to register a new major device. A PTR FAULT is injected into this function. In this case, the major device number of the new device becomes an unexpected number but the operation itself continues normally.

Untraceable: There are 11 cases in which error propagation cannot be traced completely. The faults are injected into the code for the socket management, and corrupt packet headers to be sent out to network. This example is shown in Table 5.16. The actual operations of sending out the packets are performed asynchronously. Thus, the sending-out operations cannot be traced with the kernel debugger. However, there are no noticed anything in particular on the network behavior. This is probably because the packets with incorrect headers are destroyed somewhere deeper in network drivers. As a result, this type of errors does not manifest themselves.

5.5 Summary and Discussion

This chapter investigates the Linux behavior under errors by using fault injections. In particular, this chapter focuses on the analysis on the *scope* of error propagation.

The experiments in Section 5.3.3 shows that there are chances to avoid kernel crashes in Linux by killing an offending process. As shown in Chapter 2, existing failure recoveries often track or monitor every data update despite their unaccept-

able overheads [57]. However, the concept of error propagation scope suggests that failure recoveries must track data locations, but they do not have to track data contents. Two types of error propagation scope can be distinguished by where the data is in the kernel. For example, errors in kernel stacks are process-local and ones in the heap are kernel-global. Thus, their mechanisms can be optimized by extending the crash procedure in Linux, which kills an offending process without curing the corrupted data.

The observations in Section 5.4 describes that the Linux kernel frequently checks the integrity of function arguments, return values, and other important variables. This defensive programming style probably results in low rates of global error propagation in the experiments (16% of the failures in Linux 2.6.18 and 27% of ones in Linux 2.6.38). A typical example of the defensive coding in Linux is the use of `BUG_ON` macro, which checks the integrity of the kernel internal states. The use of `BUG_ON` aids in early error detections to prevent error propagation over the entire kernel. Although it is challenging to determine if an error is kernel-global or process-local, the challenge can be mitigated by defensive coding style. One interesting direction towards more resilient Linux is to develop a systematic method that determines where `BUG_ON` macros should be inserted and the conditions given to those macros. Current static analysis tools are expected to give invaluable hints on the locations and conditions of `BUG_ON` macros.

The definition of "process-local" and "kernel-global" is somewhat ambiguous and there is room for further discussion. This is because this study does not investigate acquired resources before an injected fault is activated. For example, some errors that cause software aging can be viewed as kernel-global because a resource leakage of a process affects all the other processes in the system. On the other hand, those errors can be viewed as process-local because no global data structures are corrupted; all processes are viewing consistent image of global data structures. Other resources such as locks and interrupt states potentially cause deadlocks in kernel contexts as shown in Section 5.3.3 although they do not corrupt kernel states and can cause phenomena similar to fail-stops. In other words, the results indicate that failure recoveries with killing an offending process needs

to track acquired resources and release them during the recoveries.

Chapter 6

Conclusion

6.1 Contribution Summary

This dissertation has conducted studies of faults and errors in the Linux operating system. Natural language processing helps obtain an overview of the Linux faults by grouping 370,403 patches into 66 clusters. The study on error propagation scope indicates clear views of operating system behaviors under errors.

The clustering showed insights into fault detection in operating systems. For example, most of the faults were not surprising ones; existing tools for detecting and avoiding faults are expected to work effectively with real operating systems. On the other hand, frequent observations of low-level semantics show that testing and/or formal proof techniques with physical devices are desirable in some cases. The clusters represented the characteristics of faults in operating systems: operating system features, devices, and general software. The cluster for interrupt handling extracted 160 patches for mishandling IRQs in device drivers. This dissertation demonstrated developing static analysis with the extracted knowledge and found five faults that developers overlooked.

According to the study of error propagation scope, there are chances to avoid kernel crashes in Linux. In particular, existing failure recovery can be optimized by improving the crash procedure in Linux, which kills an offending process without curing the corrupted data. Although it is challenging to determine if or not the

failure recovery can proceed, the challenge can be mitigated by defensive coding style as observed in the in-depth analysis. The study also shows that it is necessary for the lightweight failure recovery to track acquired resources and release them during the failure recovery.

6.2 Future Directions

Although this dissertation has uncovered various characteristics of faults and errors in the Linux operating system, it only shows limited portions of their entire characteristics. For example, the clustering in the fault study did not extract rare but serious faults such as security issues. The study of error propagation did not show failures of long-running workloads.

Thus, one of the future work is to refine the methodologies to study faults and errors. In the study of faults, there can be better parameters and algorithms of natural language processing. For example, clusters that have less than 5,000 patches and more than 10,000 can be investigated. Extending the fault injector in the study of error propagation may lead to other insights that this dissertation does not show.

However, the study results suggest many directions of further researches to prevent failures in operating systems. The demonstration of static analysis developments suggests that the possibilities of automatic synthesis of static checkers from existing fault reports. The study of error propagation indicates that a tool for effectively inserting `BUG_ON` macros increases the probability of process-local errors. The kernel can be recovered from kernel-global errors if a sophisticated mechanism is developed for detecting global error propagation and releasing acquired resources.

Appendix A

Extracted Patch Documents

A.1 Example Patch Document in `memori` Cluster

commit 003615302a16579531932576bcd9582ddeba9018
Author: Johan Hovold <jhovold@gmail.com>
Date: Wed Oct 17 13:34:58 2012 +0200

USB: io_ti: fix port-data memory leak

Fix port-data memory leak by moving port data allocation and deallocation to port_probe and port_remove.

Since commit 0998d0631001288 (device-core: Ensure drvdata = NULL when no driver is bound) the port private data is no longer freed at release as it is no longer accessible.

Compile-only tested.

A.2 Example Patch Document in `irq` Cluster

commit ea6dedd7fbd0f760ebf37eb0bcc8c64856475a13
Author: Imre Deak <imre.deak@nokia.com>
Date: Mon Jun 26 16:16:00 2006 -0700

ARM: OMAP: GPIO IRQ lazy IRQ disable fix

- The current OMAP GPIO IRQ framework doesn't use the do_edge_IRQ, do_level_IRQ handlers, but instead calls do_simple_IRQ. This doesn't handle disabled interrupts properly, so drivers will still get interrupts after calling disable_irq. The patch solves this by respecting the irq_desc.disable_depth and irq_desc.running counters. When one of these is non-zero the handler is not called, the interrupt is masked and marked as pending. The pending interrupt will be serviced when the running handler returns. This is according to the same semantics as the standard do_edge_IRQ and do_level_IRQ handlers have, so one day we should use them instead of do_simple_IRQ.

- Process only interrupts that are not masked. The ISR may contain pending interrupts that are masked these shouldn't be processed.

- Move the bank IRQ unmasking out of the IRQ dispatch loop. If there are further iterations we shouldn't unmask it if there are level triggered interrupts pending.

Bibliography

[1] Amazon. *Amazon Linux AMI Security Center*. https://alas.aws.amazon.com/. Nov. 2015.

[2] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. "Virtual CPU Validation". In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. 2015, pp. 311–327.

[3] Arthur Asuncion, Max Welling, Padhraic Smyth, and Yee Whye Teh. "On Smoothing and Inference for Topic Models". In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI '09)*. 2009, pp. 27–34.

[4] M. A. Auslander, D. C. Larkin, and A. L. Scherr. "The Evolution of the MVS Operating System". In *IBM Journal of Research and Development* 25.5 (1981), pp. 471–482.

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. "Thorough Static Analysis of Device Drivers". In *Proceedings of the 1st ACM European Conference on Computer Systems 2006 (EuroSys '06)*. 2006, pp. 73–85.

[6] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. "SLAM2: Static Driver Verification with Under 4% False Alarms". In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD '10)*. 2010, pp. 35–42.

[7] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. “The Static Driver Verifier Research Platform”. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV ’10)*. 2010, pp. 119–122.

[8] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. “Composing OS Extensions Safely and Efficiently with Bascule”. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys ’13)*. 2013, pp. 239–252.

[9] Bernard Blackham, Yao Shi, and Gernot Heiser. “Improving Interrupt Response Time in a Verifiable Protected Microkernel”. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys ’12)*. 2012, pp. 323–336.

[10] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. “Latent Dirichlet Allocation”. In *Journal of Machine Learning Research* 3 (2003), pp. 993–1022.

[11] Silas Boyd-Wickizer and Nickolai Zeldovich. “Tolerating Malicious Device Drivers in Linux”. In *Proceedings of the 2010 USENIX Conference on Annual Technical Conference (USENIXATC ’10)*. 2010.

[12] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. “Parallel Symbolic Execution for Automated Real-world Software Testing”. In *Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys ’11)*. 2011, pp. 183–198.

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs”. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI ’08)*. 2008, pp. 209–224.

[14] George Candea and Armando Fox. “Crash-only Software”. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems (HotOS ’03)*. 2003, pp. 67–72.

[15] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. “Microreboot — A Technique for Cheap Recovery”. In *Proceedings of the 6th USENIX Conference on Symposium on Opearting Systems Design & Implementation (OSDI ’04)*. 2004, pp. 31–44.

[16] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. “Fast Byte-granularity Software Fault Isolation”. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. 2009, pp. 45–58.

[17] Daniel Chen, Gabriela Jacques-Silva, and Bruce Mealey. “Error Behavior Comparison of Multiple Compuing System: A Case Study Ui Linux on Pentium, Solaris on SPARC, and AIX and POWER”. In *Proceedings of the 14th IEEE Pacific Rim International Symposium On Dependable Computing (PRDC ’08)*. 2008, pp. 339–346.

[18] Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. “Security Bugs in Embedded Interpreters”. In *Proceedings of the 4th ACM Asia-Pacific Workshop on Systems (APSys ’13)*. 2013, 17:1–17:7.

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. “Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems”. In *Proceedings of the 2nd ACM Asia-Pacific Workshop on Systems (APSys ’11)*. 2011, 5:1–5:5.

[20] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. “Using Crash Hoare Logic for Certifying the FSCQ File System”. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP ’15)*. 2015, pp. 18–37.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A Platform for In-vivo Multi-path Analysis of Software Systems”. In *Proceedings of the 16th ACM International Conference on Architectural Sup-*

port for Programming Languages and Operating Systems (ASPLOS XVI). 2011, pp. 265–278.

[22] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An Empirical Study of Operating Systems Errors”. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01)*. 2001, pp. 73–88.

[23] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S. Trivedi. “Fault triggers in open-source software: An experience report”. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE ’13)*. 2013, pp. 178–187.

[24] Domenico Cotroneo, Anna Lanzaro, Roberto Natella, and Ricardo Barbosa. “Experimental Analysis of Binary-Level Software Fault Injection in Complex Software”. In *Proceedings of the 9th IEEE European Dependable Computing Conference (EDCC ’12)*. 2012, pp. 162–172.

[25] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. “Software Aging Analysis of the Linux Operating System”. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE ’10)*. 2010, pp. 71–80.

[26] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. “Remus: High Availability via Asynchronous Virtual Machine Replication”. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’08)*. 2008, pp. 161–174.

[27] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. “ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity”. In *Proceedings of the 34th ACM International Conference on Software Engineering (ICSE ’12)*. 2012, pp. 1084–1093.

[28] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. “CuriOS: Improving Reliability Through Operating System Structure”. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI ’08)*. 2008, pp. 59–72.

[29] *DDVerify*. http://www.cprover.org/ddverify/.

[30] Alex Depoutovitch and Michael Stumm. “Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes”. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys ’10)*. 2010, pp. 181–194.

[31] Joao A. Duraes and Henrique S. Madeira. “Emulation of Software Faults: A Field Data Study and a Practical Approach”. In *IEEE Transactions on Software Engineering (TSE ’06)* 32.11 (Nov. 2006), pp. 849–867.

[32] Kevin Elphinstone and Gernot Heiser. “From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?” In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP ’13)*. 2013, pp. 133–150.

[33] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. “Checking System Rules Using System-specific, Programmer-written Compiler Extensions”. In *Proceedings of the 4th USENIX Conference on Symposium on Operating System Design & Implementation (OSDI ’00)*. 2000.

[34] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. “Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code”. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP ’01)*. 2001, pp. 57–72.

[35] Micro Focus. *DFS Deutsche Flugsicherung GmbH (German Air Traffic Control)*. https://www.novell.com/success/dfs.html.

[36] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. “SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Explo-

ration". In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 2014, pp. 415–431.

[37] Archana Ganapathi, Viji Ganapathi, and David Patterson. "Windows XP Kernel Crash Analysis". In *Proceedings of the 20th USENIX Conference on Large Installation System Administration (LISA '06)*. 2006.

[38] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. "Debugging in the (Very) Large: Ten Years of Implementation and Experience". In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. 2009, pp. 103–116.

[39] Jim Gray. "Why Do Computers Stop and What Can Be Done About It?" In *Proceedings of Symposium on Reliability in Distributed Software and Database Systems*. 1986, pp. 3–12.

[40] Weining Gu, Zbingniew Kalbarczyk, and Ravishankar K. Iyer. "Error Sensitivity of the Linux kernel Executing on PowerPC G4 and Pentium 4 Processors". In *Proceedings of the 4th IEEE/IFIP International Conference on Dependable Systems and networks (DSN '04)*. 2004, pp. 887–896.

[41] Weining Gu, Zbingniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. "Characterization of Linux Kernel Behavior under Errors". In *Proceedings of the 2003 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '03)*. 2003, pp. 459–468.

[42] Lisong Guo, Julia Lawall, and Gilles Muller. "Oops! Where Did That Code Snippet Come from?" In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. 2014, pp. 52–61.

[43] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications". In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. 2011, pp. 71–83.

[44] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. “Fault Isolation for Device Drivers”. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’09)*. 2009, pp. 33–42.

[45] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. “Software rejuvenation: analysis, module and applications”. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*. 1995, pp. 381–390.

[46] Nicholas Jalbert and Westley Weimer. “Automated duplicate detection for bug tracking systems”. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’08)*. 2008, pp. 52–61.

[47] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. “Fine-grained Fault Tolerance Using Device Checkpoints”. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’13)*. 2013, pp. 473–484.

[48] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. “Tolerating Hardware Device Failures in Software”. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. 2009, pp. 59–72.

[49] Asim Kadav and Michael M. Swift. “Understanding Modern Device Drivers”. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. 2012, pp. 87–98.

[50] Wei-lun Kao, Ravishankar K. Iyer, and Dong Tang. “FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults”. In *IEEE Transactions on Software Engineering (TSE ’93)* 19.11 (Nov. 1993), pp. 1105–1118.

[51] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. 2009, pp. 207–220.

[52] E. van der Kouwe, C. Giuffrida, and A.S. Tanenbaum. "On the Soundness of Silence: Investigating Silent Failures Using Fault Injection Experiments". In *Proceedings of the 10th European Dependable Computing Conference (EDCC '14)*. 2014, pp. 118–129.

[53] Greg Kroah-Hartman. *The Linux Kernel Driver Interface*. Documentation/stable_api_nonsense.txt (in linux source tree).

[54] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-pointer Integrity". In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. 2014, pp. 147–163.

[55] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO '04)*. 2004, pp. 75–86.

[56] J.L. Lawall, J. Brunel, N. Palix, R.R. Hansen, H. Stuart, and G. Muller. "WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code". In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09)*. 2009, pp. 43–52.

[57] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. "Recovery Domains: An Organizing Principle for Recoverable Operating Systems". In *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. 2009, pp. 49–60.

[58] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. “Scaling Distributed Machine Learning with the Parameter Server”. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI ’14)*. 2014, pp. 583–598.

[59] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. “A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility”. In *Proceedings of the 2010 USENIX Conference on Annual Technical Conference (USENIXATC ’10)*. 2010, pp. 1–14.

[60] Zhenmin Li and Yuanyuan Zhou. “PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code”. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. 2005, pp. 306–315.

[61] Jacob R. Lorch, Andrew Baumann, Lisa Glendenning, Dutch T. Meyer, and Andrew Warfield. “Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-tolerant Services”. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI ’15)*. 2015, pp. 575–588.

[62] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. “A Study of Linux File System Evolution”. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST’13)*. 2013, pp. 31–44.

[63] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. 2008, pp. 329–339.

[64] Matthew Lynley. *The High Cost Of An Amazon Outage*. http://www.buzzfeed.com/mattlynley/the-high-cost-of-an-amazon-outage. Aug. 20, 2013.

[65] *Apache Mahout: Scalable machine learning and data mining*. http://mahout.apache.org/.

[66] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schuetze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[67] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Software Fault Isolation with API Integrity and Multi-principal Modules". In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. 2011, pp. 115–128.

[68] Paul D. Marinescu and George Candea. "Efficient Testing of Recovery Code Using Fault Injection". In *ACM Transactions on Computer Systems (TOCS '11)* 29.4 (Dec. 2011), 11:1–11:38.

[69] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. "A Large-Scale Study of Flash Memory Failures in the Field". In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*. 2015, pp. 177–190.

[70] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C". In *Proceedings of the 30th ACM Conference on Programming Language Design and Implementation (PLDI '09)*. 2009, pp. 245–258.

[71] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. "Representativeness analysis of injected software faults in complex software". In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*. 2010, pp. 437–446.

[72] Wee Teck Ng and Peter M. Chen. “The Systematic Improvement of Fault Tolerance in the Rio File Cache”. In *Proceedings of the 29th Symposium on Fault-Tolerant Computing (FTCS ’99)*. 1999, pp. 76–83.

[73] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. “Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs”. In *Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys ’11)*. 2011, pp. 343–356.

[74] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. “Understanding Collateral Evolution in Linux Device Drivers”. In *Proceedings of the 1st ACM European Conference on Computer Systems 2006 (EuroSys ’06)*. 2006, pp. 59–71.

[75] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. “Documenting and Automating Collateral Evolutions in Linux Device Drivers”. In *Proceedings of the 3rd ACM European Conference on Computer Systems 2008 (EuroSys ’08)*. 2008, pp. 247–260.

[76] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. “Faults in Linux: Ten Years Later”. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 2011, pp. 305–318.

[77] David A. Patterson. “An Introduction to Dependability”. In *;login;* 27.4 (Aug. 2002).

[78] Cuong Pham, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. “CloudVal: A framework for validation of virtualization environment in cloud infrastructure”. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’11)*. 2011, pp. 189–196.

[79] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. “Rethinking the Library OS from the Top Down”. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 2011, pp. 291–304.

[80] Linux Test Project. *LTP - Linux Test Project*. http://linux-test-project.github.io/. Dec. 26, 2015.

[81] LLVM Project. *LLVM Developer Policy*. http://llvm.org/docs/DeveloperPolicy.html. Dec. 26, 2015.

[82] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. “SymDrive: Testing Drivers Without Devices”. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI’12)*. 2012, pp. 279–292.

[83] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. “Detection of Duplicate Defect Reports Using Natural Language Processing”. In *Proceedings of the 29th ACM International Conference on Software Engineering (ICSE ’07)*. 2007, pp. 499–510.

[84] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. “Automatic Device Driver Synthesis with Termite”. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. 2009, pp. 73–86.

[85] Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. “User-guided Device Driver Synthesis”. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI ’14)*. 2014, pp. 661–676.

[86] Suman Saha, Jean-Pierre Lozi, Gael Thomas, Julia L. Lawall, and Gilles Muller. “Hector: Detecting Resource-Release Omission Faults in Error-handling Code for Systems Software”. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’13)*. 2013, pp. 1–12.

[87] *SLAM*. http://research.microsoft.com/en-us/projects/slam/.

[88] Mark Sullivan and Ram Chillarege. “Software defects and their impact on system availability-a study of field failures in operating systems”. In *Proceedings of the 21st IEEE International Symposium on Fault-Tolerant Computing (FTCS-21)*. 1991, pp. 2–9.

[89] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. “Membrane: Operating System Support for Restartable File Systems”. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST ’10)*. 2010, pp. 21–21.

[90] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. “Recovering Device Drivers”. In *Proceedings of the 6th USENIX Conference on Symposium on Opearting Systems Design and Implementation (OSDI ’04)*. 2004.

[91] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. “Improving the Reliability of Commodity Operating Systems”. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP ’03)*. 2003, pp. 207–222.

[92] Yuan Tian, Julia Lawall, and David Lo. “Identifying Linux Bug Fixing Patches”. In *Proceedings of the 34th ACM International Conference on Software Engineering (ICSE ’12)*. 2012, pp. 386–396.

[93] Liam Tung. *Android now has 1.4bn active users, 300m on Lollipop*. http://www.zdnet.com/article/android-has-1-4bn-active-users-with-300m-on-lollipop/. Sept. 30, 2015.

[94] Steven J. Vaughan-Nichols. *Ubuntu Linux continues to rule the cloud*. http://www.zdnet.com/article/ubuntu-linux-continues-to-rule-the-cloud/. Aug. 27, 2015.

[95] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator". In *Proceedings of the 4th ACM Annual Symposium on Cloud Computing (SOCC '13)*. 2013, 5:1–5:16.

[96] W3Cook. *W3Cook - Usage Trends, Market Share, Statistics*. http://www.w3cook.com/. Dec. 21, 2015.

[97] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. "Improving Integer Security for Systems with KINT". In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 2012, pp. 163–177.

[98] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. "Jitk: A Trustworthy In-kernel Interpreter Infrastructure". In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. 2014, pp. 33–47.

[99] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. "Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior". In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. 2013, pp. 260–275.

[100] Zhi Wang, Chiachih Wu, Michael Grace, and Xuxian Jiang. "Isolating Commodity Hosted Hypervisors with HyperLock". In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. 2012, pp. 127–140.

[101] Thomas Witkowski. "Formal Verification of Linux Device Drivers". MA thesis. TU Dresden and ETH Zurich, 2007.

[102] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. "Model Checking Concurrent Linux Device Drivers". In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. 2007, pp. 501–504.

[103] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. “Detecting Large-scale System Problems by Mining Console Logs”. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. 2009, pp. 117–132.

[104] Kazuya Yamakita, Hiroshi Yamada, and Kenji Kono. “Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery”. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’11)*. 2011, pp. 169–180.

[105] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. “Using Model Checking to Find Serious File System Errors”. In *Proceedings of the 6th USENIX Conference on Symposium on Opearting Systems Design & Implementation (OSDI ’04)*. 2004.

[106] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. “Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems”. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI ’14)*. 2014, pp. 249–265.

[107] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. “SherLog: Error Diagnosis by Connecting Clues from Run-time Logs”. In *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. 2010, pp. 143–154.

[108] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. “Be Conservative: Enhancing Failure Diagnosis with Proactive Logging”. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI ’12)*. 2012, pp. 293–306.

[109] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. “Improving Software Diagnosability via Log Enhancement”. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 2011, pp. 3–14.

[110] I.S. Zakharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, and A.V. Khoroshilov. “Configurable toolset for static verification of operating systems kernel modules”. In *Programming and Computer Software* 41.1 (2015), pp. 49–64.

[111] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. “SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques”. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’06)*. 2006, pp. 45–60.

[112] James F. Ziegler, Martin E. Nelson, James Dean Shell, R. Jerry Peterson, Carl J. Gelderloos, Hans P. Muhlfeld, and Charles J Montrose. “Cosmic ray soft error rates of 16-Mb DRAM memory chips”. In *IEEE Journal of Solid-State Circuits* 33.2 (Feb. 1998), pp. 246–252.

List of Papers

Articles on Periodicals

- Takeshi Yoshimura and Kenji Kono. "A Case for Static Analysis of Linux to Find Faults in Interrupt Request Handlers". *IPSJ Transactions on Advanced Computing Systems (ACS53)*, To appear.

- Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. "Using Fault Injection to Analyze the Scope of Error Propagation in Linux". *IPSJ Transactions on Advanced Computing Systems (ACS42)*, Vol. 6, No. 2 pp. 1-10, April 2013.

Articles on International Conference Proceedings

- Takeshi Yoshimura and Kenji Kono. "Who writes what checkers? — Learning from bug repositories". In *Proceedings of the 10th USENIX Workshop on Hot Topics in System Dependability (HotDep '14)*, pp. 1-6, October 2014.

- Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. "Is Linux Kernel Oops Useful or Not?" In *Proceedings of the 8th USENIX Workshop on Hot Topics in System Dependability (HotDep '12)*, pp. 1-6, October 2012.

- Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. "Can Linux be Rejuvenated without Reboots?" In *Proceedings of the 3rd IEEE International Workshop on Software Aging and Rejuvenation (WoSAR '11)*, pp.50-55, November 2011.