Boost the Reliability of the Linux Kernel: Debugging Kernel Oopses

Lisong Guo

Boost the Reliability of the Linux Kernel

Debugging Kernel Oopses

By

LISONG GUO

L'INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Laboratoire d'Informatique
UNIVERSITÉ PIERRE ET MARIE CURIE

A dissertation submitted to the Université Pierre et Marie Curie in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the Faculty of Informatique.

DEC 18, 2014

President of Jury:	Pierre Sens	LIP6, Paris
Rapporteur / Jury:	Nicolas Anquetil	Inria, Lille
Rapporteur / Jury:	Laurent Réveillère	LaBRI, Bordeaux
Jury member:	Nicolas Palix	IMAG, Grenoble
Jury member:	David Lo	SMU, Singapore
Directrice / Jury:	Julia Lawall	LIP6, Paris
Directeur / Jury:	Gilles Muller	LIP6, Paris

Abstract

When a failure occurs in the Linux kernel, the kernel emits an error report called *"kernel oops"*, summarizing the execution context of the failure. Kernel oopses describe real Linux errors, and thus can help prioritize debugging efforts and motivate the design of tools to improve the reliability of Linux code. Nevertheless, the information is only meaningful if it is representative and can be interpreted correctly.

In this thesis, we study a collection of kernel oopses over a period of 8 months from a repository that is maintained by Red Hat. We consider the overall features of the data, the degree to which the data reflects other information about Linux, and the interpretation of features that may be relevant to reliability. We find that the data correlates well with other information about Linux, but that it suffers from duplicate and missing information. We furthermore identify some potential pitfalls in studying features such as the sources of common faults and common failing applications.

Furthermore, a kernel oops provides valuable first-hand information for a Linux kernel maintainer to conduct postmortem debugging, since it logs the status of the Linux kernel at the time of a crash. However, debugging based on only the information in a kernel oops is difficult. To help developers with debugging, we devised a solution to derive the *offending line* from a kernel oops, *i.e.*, the line of source code that incurs the crash. For this, we propose a novel algorithm based on approximate sequence matching, as used in bioinformatics, to automatically pinpoint the offending line based on information about nearby machine-code instructions, as found in a kernel oops. Our algorithm achieves 92% accuracy compared to 26% for the traditional approach of using only the oops instruction pointer. We integrated the solution into a tool named **Oopsa**, which would relieve some burden for the developers with the kernel oops debugging.

RÉSUMÉ

Lorsqu'une erreur survient dans le noyau Linux, celui-ci émet un rapport d'erreur appelé *"kernel oops"* contenant le contexte déxécution de cette erreur. Les kernel oops décrivent des erreurs réelles de Linux, permettent de classer les efforts de débogage par ordre de priorité et de motiver la conception d'outils permettant d'améliorer la fiabilité du code de Linux. Néanmoins, les informations contenues dans un kernel oops n'ont de sens que si elles sont représentatives et qu'elles peuvent être interprétées correctement.

Dans cette thèse, nous étudions une collection de kernel oops provenant d'un dépôt maintenu par Red Hat sur une période de huit mois. Nous considérons l'ensemble des caractéristiques de ces données, dans quelle mesure ces données reflètent d'autres informations à propos de Linux et l'interprétation des caractéristiques pouvant être pertinentes pour la fiabilité de Linux. Nous constatons que ces données sont bien corrélées à d'autres informations à propos de Linux, cependant, elles souffrent parfois de problèmes de duplication et de manque d'informations. Nous identifions également quelques pièges potentiels lors de l'étude des fonctionnalités, telles que les causes d'erreurs fréquentes et les causes d'applications défaillant fréquemment.

En outre, un kernel oops fournit des informations précieuses et de premiŕe main pour un mainteneur du noyau Linux lui permettant d'effectuer le débogage post-mortem car il enregistre l'état du noyau Linux au moment du crash. Cependant, le débogage sur la seule base des informations contenues dans un kernel oops est difficile. Pour aider les développeurs avec le débogage, nous avons conçu une solution afin d'obtenir la ligne fautive à partir d'un kernel oops, *i.e.*, la ligne du code source qui provoque l'erreur. Pour cela, nous proposons un nouvel algorithme basé sur la correspondance de séquences approximative utilisé dans le domaine de bioinformatique. Cet algorithme permet de localiser automatiquement la ligne fautive en se basant sur le code machine à proximité de celle-ci et inclus dans un kernel oops. Notre algorithme atteint 92% de précision comparé à 26% pour l'approche traditionnelle utilisant le débogueur gdb. Nous avons intégré notre solution dans un outil nommé OOPSA qui peut ainsi all eger le fardeau pour les développeurs lors du débogage de kernel oops.

DEDICATION AND ACKNOWLEDGEMENTS

The thesis is a dream. I have been indulged with a great expectation from my parents and my siblings about my career, ever since when I was a kid. The seed of pursuing a doctor degree had been planted in my mind since then. The idea, eventually became my own dream and had been developed during my school time. Here, first, I would like to dedicate this thesis to my family in response to their expectation.

The thesis is a chance in a lifetime. Looking back at the 18 years that I had in school before I started my thesis, I cannot say that I had never had a doubt that one day I would pursue a PhD degree. I just did what I can do the best, without staring at the goal which seemed to be too far to see at that time. It is my schoolmates, teachers and professors who believed in my potential, and who gave me numerous supports and help, put me back to the track when I tumbled and distracted. I would like to dedicate this thesis to all of them.

The thesis is a grand mission. I have been granted with a funding, a time limit, and two supervisors to accomplish the thesis. It would have been a mission impossible, without the guidance and the extremely patient review from my supervisors Julia Lawall and Gilles Muller. It was them who introduced me into the academic world, and who showed me how to conduct research, how to write a good paper and how to present a work. I hold a great respect and gratitude for them.

The thesis is an adventure. Literally, I have travelled quite a lot across France and Europe during my thesis. Most of them are generally sponsored by Inria for me to attend conferences, winter and spring schools. Also, I ventured plenty of trips myself. During all these travels, I have meet many people who made my experience wonderful and interesting. Particularly, I would like to thank Prof. Tien Nguyen who invited me to Iowa State University to work on a collaboration project. I had a great time sharing the office with Jafar, Robert and Amine during my short stay. All these experiences have greatly expanded my view about Europe and USA. I am grateful to all the people who made this possible.

The thesis is a grind work. A thesis in *Informatique* involves a lot of time sitting in the office and staring at the screen. It would have been extremely boring without the accompany of all the

colleagues in the office. Thanks to Mahsa, Peter, Marek, Masoud, Pierpaolo and Florian, with whom we had numerous fun conversations during lunch and coffee breaks.

The thesis is another exam. Before the moment I started my thesis, I have went through numerous written exams in my life. Yet, the thesis is the longest and the most challenging one I faced so far. It is also the most unusual one, for which I need to come up the question myself and attempt an answer. Before I stand for my defense, I would like to thank all my juries (Pierre Sens, Laurent Réveillère, Nicolas Anquetil, Nicolas Palix, David Lo, Julia Lawall and Gilles Muller) for letting me have the honour of having them as my juries. More over, I am really grateful for their efforts on reviewing my work.

The thesis is piles of paper work. Not just all the papers, slides and posters that I prepared, I have been through a tremendous amount of paper works more than I can imagine, starting from before my arriving in France. Here I would like to thank all the secretaries (Stéphanie Chaix, Hélèn Milome, Cecile Bertrand-Kalkofen, Martine Girardot and Marguerite Sos) who helped me going through all those processes patiently.

The thesis is a long story. I met numerous people and had numerous stories during this thesis journey. I could make another thesis if I list all of the stories. For now on, they would stay in my mind, and one day I would share them my kids and grand kids.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Starting from the general background of our work, in this beginning chapter, we illustrate the research problems that this thesis is trying to resolve, and we then list the contributions that we made in this attempt. We conclude this chapter with a brief note on how this thesis is organized.

1.1 Motivation

The Linux kernel is used today in environments ranging from embedded systems to servers. It resides in the core of numerous distributions of operating systems, such as Android, Debian and Fedora, *etc.* While the Linux kernel is widely regarded as being stable for ordinary use, it nevertheless still contains *bugs* [103], *i.e.* defects in the software. The Linux kernel is inevitably buggy for three major reasons. First, it consists of a huge code base, *i.e.* over 15 million lines of code, and continues to grow [33]. It is extremely challenging to ensure correctness on such a huge code base. Second, the development and maintenance of the Linux kernel are highly dynamic. There is a major release of the Linux kernel every 3 months, to which over a thousand developers all over the world contribute. The introduction of new code as well as the fixes of existing bugs bring new bugs to the kernel. Third, the Linux kernel offers a huge variety of services (devices drivers, file systems, etc.), and each of them comes with a range of configuration options, which results in a huge number of configuration settings for the Linux kernel [119]. Therefore, it is essentially impossible to exhaustively test the Linux kernel for all possible configurations. Due to these reasons, it is impossible to eliminate bugs from the Linux kernel. As long as the Linux kernel evolves, the bugs would persist, but at a tolerable rate thanks to the continuous engineering effort.

Other than the measures that prevent bugs slipping into the Linux kernel as much as possible, Linux kernel developers often resort to the ***postmortem debugging***. Debugging is the process of localizing bugs in software, finding their root causes and providing fixes for them. Postmortem debugging refers to the approach of debugging a bug after its manifestation, rather than predicting the bug.

The manifestation of a bug involves some *errors*, which are unusual states in the software. Since a bug is often hard to reproduce, it is critical to catch and document errors for later debugging. In order to facilitate debugging, developers commonly equip software with error-reporting capacity, so that the software can automatically generate information about errors when they occur. The generated error reports are then collected and later serve as an important starting point in finding bugs.

Following the same practice, Linux kernel developers have designed the kernel to generate a specific kind of error report, called a *kernel oops*, whenever the kernel crashes. Between 2007 and 2010, kernel oopses were collected in a repository under the supervision of `kernel.org`, the main website for distributing the Linux kernel source code. There used to be a mailing list [125] that was aimed at discussing and addressing the most frequent kernel oopses. Similarly, the practice of maintaining and debugging error reports has also been adopted in Windows [97] and Mac OS X [1]. Error reports from Windows have been used for tasks such as prioritizing debugging efforts, detecting the emergence of malware, and monitoring the resurgence of bugs [44]. Indeed, the data in such repositories has the potential to help developers and software researchers alike identify the important trouble spots in the software implementation.

In this thesis, we study error reports in the context of the Linux kernel, *i.e.* kernel oopses. The critical nature of many of usage scenarios of the Linux kernel means that it is important to understand and quickly address the errors in the Linux kernel that are encountered by real users. In this thesis, we intend to approach the above two goals respectively with a systematic study on a large collection of kernel oopses and a tool to help developers better debug kernel oopses.

1.2 Research Problem

Our objects of study are kernel oopses, which are a specific kind of error reports. A kernel oops is emitted from the Linux kernel, at the time of crash. Given a kernel oops, kernel developers need to understand the information contained in the kernel oops, and then find the problem that triggered the kernel oops. Therefore, concerning kernel oopses, our research problem concerns two aspects: ***kernel oops comprehension*** and ***kernel oops debugging***. In brief, kernel oops comprehension concerns understanding of the information carried by a kernel oops, as well as the knowledge that can be extracted from a collection of kernel oopses, while kernel oops debugging focuses on the actions that one can take towards finding and fixing the problem that triggers the kernel oops. We discuss each aspect in detail in the following sections.

1.2.1 Kernel Oops Comprehension

A kernel oops holds key information about the status of the Linux kernel during the crash. A kernel oops is designed to be concise, unlike another similar error report, *e.g.* the kernel dump [111], which is a snapshot of data in the memory and whose size ranges from several hundred MBs to several GBs. The conciseness of kernel oops facilitates its transfer. But inevitably there is certain information that is skipped in a kernel oops, which requires developers to recover and derive this information from the content of the kernel oops.

Kernel oopses are generated automatically in the Linux deployments at the users' sites. But the collection of kernel oops is voluntary with the authorization of users. Since September 2012, Red Hat has revived the collection of Linux kernel oopses in the repository at `oops.kernel.org` [12], which receives oops reports from any Linux distribution. This repository reopens the possibility of using kernel oopses to understand the real errors encountered by Linux kernel users. Nevertheless, extracting knowledge from a collection of kernel oopses is difficult. The Linux kernel has a number of properties that make interpreting kernel oopses challenging.

The first major challenge in interpreting the Linux kernel oops is that of ***origin obscurity***. One of the most critical pieces of information about a kernel oops is its origin, *i.e.* the source code that generates the kernel oops and the hardware platform that runs the source code. However, it is hard, if not impossible, to accurately identify the origin of a kernel oops. The Linux kernel is highly configurable, exists in many versions, and is packaged into many distributions such as Fedora, Debian, Android, *etc*. Within a kernel oops, this version information is poorly documented. Even given the exact version number, it is not easy to associate the source code with the kernel oops. Because each distribution can customise the Linux kernel, it is hard to track all the customisations. As to the other part of origin information, the hardware platform, it is not necessarily present in a kernel oops. The Linux kernel runs on a vast number of hardware platforms, among which there are many proprietary ones where the companies would not bother to put their platform information in a kernel oops. Indeed, we observed quite a lot kernel oopses with the platform information stated as `To be filled`. We suspect that this is partially due to privacy concerns, as many companies are not willing to reveal too much information in an error report even they agree to submit the report for trouble shooting. Or it is simply because the administrators of the system do not bother to fill the field.

The second major challenge in interpreting the Linux kernel oops is the issue of ***format inconsistency***. First, kernel oopses are generated in an ad-hoc manner, *i.e.* the structure and the content of a kernel oops depend on the type of error that triggered it. For instance, a NULL-pointer dereference error generates a kernel oops that contains a machine code snippet around the memory address referred by the instruction pointer, but this code snippet is not present in kernel oopses that are triggered by some hardware related errors. Second, an oops can be transferred in different ways by different Linux distributions. For security and performance

reasons, the Linux kernel does not itself submit kernel oopses to the repository, but only have them written to a kernel log file. Different Linux distributions provide their own user-level applications for retrieving oopses from these log files and submitting them to the repository. These tools might decide to skip certain information in a kernel oops. For instance, we noticed that recent versions of Fedora remove the random ID within a kernel oops during submission.

Due to the above challenges, it is difficult to extract all the necessary information from kernel oopses and draw reliable conclusion about the information. In Chapter 3, we conduct a systematic study on a recent repository of kernel oopses. In addition, we detail the challenges that we encountered and the solutions we proposed.

1.2.2 Kernel Oops Debugging

A collection of kernel oopses as a whole provides some insight about the Linux kernel. On the other hand, an individual kernel oops has the potential to help kernel developers with their daily work. The main tasks of a kernel developer include adding new functionalities to the Linux kernel, as well as addressing the problems that have been identified in the implementation of existing functionalities. A kernel oops is useful in the latter task. Indeed, as a crash report, a kernel oops documents an instantaneous image of the kernel state at the time of the crash. Therefore, a kernel oops provides first-hand information about the manifestation of a bug in the Linux kernel, which is essential for addressing the bug later. At the same time, a kernel oops is logged primarily at the level of machine code, which makes it difficult to interpret, drastically limiting its practical benefit. Thus, it is desirable for kernel developers to have techniques or tools that can help them with the debugging of individual kernel oopses.

A fundamental part of debugging any software crash including a kernel oops is to identify the *offending line*, *i.e.*, the line of the source code at which the crash occurs. To illustrate the importance of knowing the offending line and the difficulty of obtaining it from a kernel oops, even in a favourable case when it is possible to interact with the user who encountered the crash, we describe a real case [28] from the Linux kernel bugzilla. A user named Jochen encountered frequent crashes on his XFS file system and filed a bug report that included a kernel oops. A developer named Dave took charge of the report, and started to debug the issue. As Dave did not have access to the compiled kernel that Jochen had used, he instructed Jochen to go through a series of procedures on the victim machine in order to produce the offending line information. In all, it took 16 comment exchanges before another user, Katharine, who had the same problem, managed to obtain the offending line. Dave then quickly figured out the source of the bug and provided a corresponding patch.

One solution to get the offending line information would be to instrument the Linux kernel to include the offending line number in the kernel oops. Yet, this is not a viable solution. First of all, the strategy is intrusive, which requires an update of the Linux kernel. Since some users have their own customised Linux kernel, an update to the kernel might require users to recompile

and reload the kernel. This requirement could hinder the adoption of the solution. Secondly, the instrumentation would introduce an overhead to the Linux kernel runtime, in addition to inflating the Linux kernel image size. Indeed, a recent proposal (September, 2012) [70] to extend the Linux kernel to make it possible to include line number information was rejected, as it would incur a too great (albeit small) space overhead (10MB) in a running kernel. Last and most importantly, the solution does not solve the problem of debugging the large number of reports already available in the kernel oops repository, which possess a potential to boost the reliability of the Linux kernel if we could make use of them.

Therefore, a desirable solution for the problem of pinpointing the offending line of a kernel oops would be having an automatic tool that can locate the offending line, given an existing kernel oops, without any human intervention or system instrumentation.

1.3 Contribution

We conduct our work with regards to the two aspects of our research problem described above. In this section, we list the contributions that we have made in resolving the problem.

Kernel oops comprehension. We study how we can interpret Linux kernel oopses to draw conclusions about Linux kernel reliability. To this end, we perform a study of over 187,000 Linux kernel oopses, collected in the repository maintained by Red Hat between September 2012 and April 2013. We first study properties of the data itself, then correlate properties of the data with independently known information about the state of the Linux kernel, and then consider some features of the data that can be relevant to understanding the kinds of errors encountered by real users and how these features can be accurately interpreted. The main lessons learned are as follows:

- The number of oopses available in the repository for different versions varies widely, depending on which Linux distributions have adopted the version and the strategies used by the associated oops submission tools.
- The repository may suffer from duplicate and missing oopses, since the available information does not permit identifying either accurately.
- Identifying the service causing a kernel crash may require considering the call stack, to avoid merging oopses triggered in generic utility functions. The call stack, however, may contain stale information, due to kernel compilation options.
- Analyses of the complete stack must take into account that the kernel maintains multiple stacks, only one of which reflects the current process's or interrupt's execution history.

- Kernel executions may become tainted, indicating the presence of a suspicious action in the kernel's execution history. Oopses from tainted kernels may not reflect independent problems in the kernel code.

Kernel oops debugging. We help kernel developers to debug a kernel oops through pinpointing the offending line of the kernel oops. To this end, we propose a novel algorithm based on approximate sequence matching, as used in bioinformatics, to automatically pinpoint the offending line based on information about nearby machine-code instructions as found in a kernel oops. We further integrate our approach into an automatic tool named **OOPSA** that takes only the kernel oops as input and produces the offending line number.

To evaluate OOPSA, we conducted a series of experiments on 100 randomly selected examples from the kernel oops repository and the Linux kernel bugzilla. OOPSA achieves 92% accuracy in pinpointing the offending line, compared to the 26% accuracy obtained through the state-of-the-art approach. In addition, our algorithm has the linear complexity in the size of the crashing function. For the kernel oopses that we have tested, it takes on average 31ms to produce the result on a current commodity machine. Therefore, our solution is effective and efficient, and thus it can ease the burden on a Linux kernel developer in debugging a kernel oops.

In resolving this problem, we have made the following contributions:

- We quantify the difficulty of finding the offending line in an oops generated by an older kernel.
- We propose and formalize an algorithm using approximate sequence matching for identifying the line of source code that led to the generation of an oops.
- We show that the complexity of our algorithm is linear in the size of the crashing function.
- We show that our approach is effective, achieving an accuracy of 92% on 100 randomly selected examples.

1.4 Organization

The rest of this thesis is organized as follows: Chapter 2 gives literature reviews in the domain of this thesis. Chapter 3 addresses the aspect of kernel oops comprehension, with regard to the kernel oops repository. Chapter 4 discusses our solution to pinpoint the offending line of a kernel oops, which concerns the aspect of kernel oops debugging. Chapter 5 discuss some work that is related to the techniques that we devised in Chapter 4. Chapter 6 concludes this thesis.

CHAPTER 2

BACKGROUND

In this chapter, we give a review of the literature on the domains that are related to our work. We focus on improving the reliability of the Linux kernel through helping Linux kernel developers better debug the faults that are encountered by users. Overall, our work falls into the domain of ***software debugging***. More specifically, our work can be further classified as one of the techniques of ***software fault localization*** which is one of the most critical activities that are involved in software debugging. We developed a technique to help developers pinpoint the offending line of a kernel oops. The technique is inspired by the ***sequence alignment*** algorithms which were initially proposed in bioinformatics domain. In this chapter, we review these three subjects to serve as the background knowledge of this thesis.

2.1 Software Debugging

Software bugs are defects that reside in software and cause software to behave abnormally. The abnormal behaviours of software lead to economic loss and damage. A public report [45] from NIST (US National Institute of Standards and Technology) in 2002 stated that software bugs cost the US economy 59.5 billion dollars annually. Therefore, it is of great interest for developers to reduce software bugs during the software development process.

The process of locating and fixing defects in software is called software debugging. It takes developers a significant amount of time and resources during the development of software. It is estimated that debugging activities account for 30%~90% of labor expended for a software project [24]. Therefore, techniques and tools that can help developers with software debugging tasks can have a significant impact on the cost and quality of software.

Software debugging is a broad topic that covers various software artefacts such as source code, bug and error reports, *etc.* Concerning all the activities of software debugging, there have been

a number of works [57, 58, 80, 128] dedicated to *bug reproduction*, which aim to produce a bug in a repeatable manner in order to help developers identify the key factors that trigger the bug. Nevertheless, bug reproduction is expensive and most of the time it requires the involvement of end users or full control over the execution environment. Given the above restriction, it is thus a challenge for developers to conduct bug reproduction in real world. We focus on the forensic analysis, particularly error report debugging which aims to diagnose and analyse the manifestation of bugs, instead of reproducing bugs. As the background knowledge, we review *bug reproduction* and *error report debugging* in the rest of this section.

2.1.1 Bug Reproduction

Being able to reproduce a bug is crucial for debugging, especially for concurrency bugs, yet it is notoriously difficult due to non-deterministic factors such as program inputs, memory status, and thread scheduling. A spectrum of techniques are proposed to address the bug reproduction problem, ranging from record-replay techniques [57, 58, 79, 96, 126] that capture the non-deterministic information online, to execution synthesis techniques [128, 133] that rely on offline simulation and analysis. Several hybrid techniques [80, 81, 104, 137] attempt to explore the balance between online recording and offline searching. To demonstrate the idea of bug reproduction and its limitations, we review in detail a recent work [59] about reproducing concurrency bugs as follows:

CLAP: Recording Local Executions to Reproduce Concurrency Failures. Huang *et al.* [59] proposed the technique **CLAP** to reproduce concurrency bugs. CLAP consists of two key steps. First, it logs the local execution paths of threads at runtime, which is called *online* processing. Second, after the manifestation of a current bug, it constructs and synthesizes constraints that are necessary for the bug to manifest, including both intra- and inter-thread constraints, and then it uses a constraint solver to produce a global schedule that respects all the constraints. The step is also referred as *offline* processing. CLAP has several advantages. The main advantage of CLAP is that its logging is substantially cheaper than that performed by previous solutions [80, 81, 137], which log the memory status of the program, or which add memory barriers for synchronization [58, 79, 137]. Therefore, it minimises the perturbation to the runtime, and thus reduces the possibility of not being able to reproduce the bug due to the extra overhead caused by the perturbation. Secondly, CLAP works on a range of relaxed memory models, such as Total Sequential Order (TSO) and Partial Sequential Order (PSO) [122], in addition to sequential consistency. Finally, CLAP can parallelize constraint solving, which mitigates the scalability issues.

CLAP is implemented on top of LLVM [78] and KLEE-2.9 [29], with the STP constraint solver [42]. It adopts the Ball-Larus [20] path profiling algorithm to log the paths for multi-threaded `C/C++` programs, which incurs 9.3%-269% runtime overhead. CLAP captures all the

necessary execution constraints of concurrent programs with the formula $\Phi = \Phi_{path} \wedge \Phi_{bug} \wedge \Phi_{so} \wedge \Phi_{rw} \wedge \Phi_{mo}$, where

- Φ_{path} (***Path Constraints***): The path constraints are derived from the symbolic execution of the logged paths of thread execution, *i.e.* a conjunction of constraints that specify the conditions of each branch taken by the threads.
- Φ_{bug} (***Bug Manifestation Constraints***): The bug manifestation constraints are expressed as predicates over the final program state. For example, a NULL pointer dereference error such as a –> b can be defined as $V_a = NULL$.
- Φ_{so} (***Synthronization Order Constraint***): The synchronization order constraints consist of two types of constraints that are dictated by the synchronization operations, *i.e.* the *locking constraints* enforced by the `lock/unlock` operations, and the *partial order constraints* implied by the `fork/join` and `wait/signal` operations.
- Φ_{rw} (***Read-Write Constraints***): The Read-Write constraints state that the value of a `Read` operation should be associated with a most recent `Write` operation.
- Φ_{bug} (***Memory Order Constraints***): The memory constraints express the memory model of a program, such as TSO, PSO and sequential consistent.

The above constraints are complete, with regards to all concurrency situations, because the intra-thread data and control-flow dependencies are captured by Φ_{path} and Φ_{bug} constraints, and the inter-thread dependencies are captured by Φ_{so}, Φ_{rw} and Φ_{mo} constraints. In addition, CLAP also models the number of context switching during the thread scheduling as a constraint. This allows it to reproduce a bug with a schedule of least context switching, which is more realistic for the manifestation of a concurrency bug, since the more context switching that it requires, the less likely the scheduler runs into the case and the less likely that a bug will appear.

CLAP can be considered as a part of the state-of-the-art in bug production, particularly for concurrency bugs which are the most difficult ones to reproduce. Yet, despite of all its advancement, the techniques of bug reproduction intrinsically impose a strong restriction on its usage scenario, *i.e.* an instrumented software run in a controlled execution. This restriction can only be met by the consensus and the cooperation of users, if the software is deployed in real world, or in an in-house testing environment on which developers have the full control.

In the case of the Linux kernel debugging, typically the Linux kernel developers have no access to the environment of users. Therefore, bug reproduction is not realistic in this common case. Instead, kernel developers more often resort to *error report debugging*, *i.e.* they receive error reports that are generated and collected on the machines, without the user intervention, and start debugging from that point on. The entire debugging process has no input or at least the minimal feedbacks from users. We fit our work into this scenario, and devise our solution bearing this assumption. Next, we review some related work on error report debugging.

2.1.2 Error Report Debugging

An error report is a text message generated by software at the time of failure. The message contains some predefined strings, and more importantly, the values of some parameters concerning the running status of software. An error report serves two purposes: First, it notices users about the status of a running software; Second, it provides an important piece of information for developers to conduct software debugging. Out of the first purpose, the capability of generating error report is necessary for all software. And it is also practical to have the capability for both users and developers. Not surprisingly, the practise of generating, collecting and debugging error reports is universally adopted in all operating systems, *e.g.* Windows, Macintosh and Linux.

Windows Error Reporting (WER) [44, 97] is a post-mortem error debugging system that collects and processes error reports generated by the Windows operating system as well as over 7000 third-party applications from a billion machines. WER consists of both a client and a server service. The client service runs on users' machines, monitoring the status of Windows operating system as well as applications on top of the system. Once an error occurs, the client service intercepts and reports the error message to the server service. Before sending, the error message is hashed and compressed, in order to avoid duplicated reports and to reduce the traffic load.

Due to the wide adoption of Windows system, WER receives millions of reports daily, which requires enormous engineering force to be able to fully take care of. Thus, one of the primary goals of WER is to help programmers prioritize and debug the most urgent issues reported from users. For this, WER groups the error reports into buckets based on a series of criteria, such as the crashing program name, the exception code, and the trapping instruction offset. Ideally, the error reports within a bucket correspond to a specific bug, or at least correlated bugs. The strategy of bucketing is essential to WER, as it helps developers concentrate their debugging effort. Later, a tool named ReBucket [35] proposed another method of bucketing for WER, which is to cluster crash reports based on call trace.

Similarly, on the Linux platforms, there is a kind of error report, called *kernel oops*, which is generated during the Linux kernel crash. Unlike WER, there is no systematic way on the Linux platforms to gather and analyze kernel oopses. Kernel oopses are kept on log files in the client machine. Each Linux distribution (*e.g.* Debian, Fedora) has its own tool set to extract and submit kernel oopses. In addition, each distribution has its own repository to collect kernel oopses.

It is not so intuitive to collect a large amount of error reports from the Linux kernel for study. One of the reasons is that the Linux kernel is relatively stable, as the quality of the Linux kernel code has improved significantly, *e.g.,* the number of faults per line has dropped over the past 20 years [32, 103]. It is relatively rare, if not unlikely, to observe the crash of the Linux kernel. Some researchers then applied a technique called fault injection to artificially inject faults into the software system. For instance, Gu *et al.* [46] performed a series of fault injection experiments on the Linux kernel of version 2.4.20, to simulate the Linux kernel crash. Their goal was to analyze and quantify the response of the Linux kernel in various crashing scenarios. Most

recently, Yoshimura *et al.* [132] did an empirical study on the Linux kernel as well with fault injection. Their goal was to quantify the damage that could be incurred at the crash time of the Linux kernel.

Other than above work, to our knowledge, there is few work that studies kernel oopses from the real-world, rather than the ones generated through fault injection. Since September 2013, the kernel oops repository (`oops.kernel.org`) [12] that can receive kernel oops reports from all Linux deployments has been revived by Red Hat. The repository opens a new opportunity for a systematic study of kernel oopses. We intend to fill this gap with this thesis.

2.2 Software Fault Localization

This thesis addresses the issue of debugging the Linux kernel oopses, which ultimately is to help kernel developers quickly locate the faults in Linux kernel that cause the kernel oopses. Therefore, in this section, we cover the general background of software fault localization, and present some software fault localization techniques. First, we classify the basic and essential terminology [14] in this domain as follows:

Definition 2.1 (*Failure*). A *failure* is an event that occurs when the delivered service deviated from the correct service.

Definition 2.2 (*Error*). An *error* is a system state that may cause a failure.

Definition 2.3 (*Fault*). A *fault* is the cause of an error in the system.

A failure is the phenomenon observed by the users. A fault is the defect residing in the source code that eventually leads to the failure of the system. During the manifestation of a failure, the system encounters some errors.

We illustrate the relationship between these three concepts in the following figure. The symbol $\dashrightarrow$ indicates that the causal relationship, *i.e.* the concept on the left hand side triggers the occurrence of the concept on the right hand side. But the relationship is NOT strong, which is why we put it in dash, *i.e.* the presence of the left one does not always imply the occurrence of the right one. For instance, the existence of faults in a software system does not always render the system into failure. Most of the failures only manifest in certain corner cases.

$$Fault \dashrightarrow Error \dashrightarrow Failure$$

Software fault localization is the activity of identifying the exact locations of program faults in source code [131]. DeMillo *et al.* [38] analysed the debugging process and observed that developers consistently perform four tasks when attempting to locate faults in a program: 1). identify statements involved in failures – those executed by failed test cases; 2). narrow the search by selecting suspicious statement that might contain faults; 3). hypothesise about suspicious faults 4). and restore program variables to a specific state.

Most of the works about software fault localization, as we list in Table 2.1, are concerned with the second task above. They are often built on the assumption that there are abundant test cases, and one can obtain the code coverage information after the execution of test cases. The code coverage includes the information about whether a statement or a block of statements is involved in certain run of execution. The collection of code coverage information is also the first task of fault localization, as listed above, and is often achieved by instrumentation [4]. However, there is little work concerning about the first task of fault localization. In this thesis, we address this task, in terms of pinpointing the *offending line* of kernel oops, in the context of Linux kernel debugging.

In the rest of this section, we review the techniques of software fault localization presented in Table 2.1.

Table 2.1: A general classification of fault localization techniques [91]

techniques	references
spectrum-based	Tarantula [64], Ochiai [5–7], Pinpoint [30]
program slicing	slicing survey [120], program slicing [130], dynamic slicing [10, 134, 135]

2.2.1 Spectrum-Based Fault Localization

We first explain the basic idea behind spectrum-based localisation techniques. As examples, we then review two state-of-the-art techniques (Tarantula [64], Ochiai [5]) in detail.

Definition 2.4 (*Program Element*). The source code of a program P can be represented as a set of basic elements, where each element e_i can be a statement, a block from the control flow graph, a function, a component, etc., *i.e.* $P = \{e_1, e_2, ...e_n\}$.

Definition 2.5 (*Faulty Element*). A *faulty element* f is an element from a faulty program P, *i.e.* $f \in P$, where the fault resides.

Definition 2.6 (*Program Spectrum*). A *program spectrum* [109] S is a mapping between a set of program elements denoted as $E = \{e_1, e_2, ...e_n\}$ and the result of a specific execution of the program (*i.e.* a passed or failed test case) denoted as R, *i.e.* $S = \langle E, R \rangle$.

A program spectrum is usually generated and collected after the execution of a test case. It can then be viewed as an execution trace of program, which represents the behaviours of the program under a certain circumstance. A collection of program spectra can give a global view of the behaviour of a program.

We give an example of a kind program spectrum called a *program hit spectrum* in Figure 2.1. On the left side, there is a matrix where each row corresponds to the involvement of each program element in an execution. The values in each column i represent the involvement of the element

e_i in the executions. For instance, the value at the starting position (1, 1) of the matrix is 1, which says that element e_1 is involved in the execution that produces E_1, while the value to its right at (1, 2) indicates that element e_2 is NOT involved in the execution that produces E_1. On the right side of Figure 2.1, there is a vector that represents the results (passed or failed) of each execution. For example, the value R_1 is 0, which indicates that the execution that produces E_1 passes without any error. While the value R_4 is 1, which indicates the the execution that produces E_4 fails. There are other forms of program spectra, which one can find in the empirical study conducted by Harrold *et al.* [52].

$$\begin{array}{c} \\ E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \end{array} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \Longleftrightarrow \begin{array}{c} \\ R_1 \\ R_2 \\ R_3 \\ R_4 \\ R_5 \end{array} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Figure 2.1: An example of program spectrum

Definition 2.7 (*Suspiciousness Score*). Given a faulty program $P = \{e_1, e_2, ...e_n\}$ and a collection of program spectra $T = \{S_1, S_2, ...S_n\}$, we then define $\omega(e_i, T)$ as a *suspiciousness score* function that gives a quantitative score of the likelihood that the element e_i is a faulty element, given the circumstances specified by T.

Following the example of the program spectrum in Figure 2.1, we give an example of suspiciousness scores in Figure 2.2. Given the collection of program spectra $\langle E_i, R_i \rangle$, we assign a suspiciousness score for each element e_i (listed in the bottom row of the table), based on the suspiciousness score function in the tool Tarantula [64] which we would discuss later. Among all the elements, element e_4 is the most suspicious one. As one can see from the figure, the involvement of element e_4 is the most consistent with the result of the execution. Especially, if the element e_4 is involved, then the execution most likely fails. Otherwise, the execution pass.

E	e_1	e_2	e_3	e_4	e_5	R
E_1	1	0	0	0	0	0
E_2	1	1	0	0	0	0
E_3	1	1	1	1	1	0
E_4	1	1	1	1	1	1
E_5	1	1	1	0	1	0
ω	0.50	0.56	0.62	0.71	0.50	

Figure 2.2: An example of suspiciousness score

Spectrum-based Fault Localization. Given the above definitions, spectrum-based fault localization techniques aim to find a *good* suspiciousness score function that can identify true

faulty elements with high suspiciousness values. A good suspiciousness score function ranks the true faulty elements on top and filters out the non-faulty elements, so that it can help developers to quickly locate the fault and thus accelerate the debugging process.

The general approach of spectrum-based fault localization often employs statistical analysis on the association of each program element with execution failures, based on the intuition that the stronger the association, the more likely the fault is located in the element. We now present in detail two state-of-the-art tools, named Tarantula [64] and Ochiai [5], that follow this approach.

Tarantula. Jones *et al.* [64] presented a tool named *Tarantula* to assist developers locating faulty statements within a program. The fundamental idea of Tarantula is to create a visualisation of the source code in which individual statements are coloured according to their participation in a given test suite. Tarantula is unique in its colouring schema, compared to other similar solutions [9, 18]. The colouring schema of Tarantula consists of two components: ***colour*** and ***brightness***. Its colour range spreads across the entire spectrum from red to yellow, and from yellow to green. At one end, the red colour is used to mark the statements that are only involved in the failed test cases. At the other end, the green colour marks the statements that are only involved in the passed test cases. In the middle of the spectrum, the yellow colour marks the statements that are involved in both the passed and failed test cases. The more a statement is involved in failed test cases, *i.e.* the higher the percentage of failed test cases that cover the statement, the more reddish the statement is marked, and vice versa for the green colouring. The more a statement is involved in failed test cases, the brighter is its colour. The authors refer to their solution as a *continuous* approach, as opposite to the *discrete* colouring schema [9] that marks the statements with only three basic colours (red, yellow and green) regardless the degree of their participation in the outcome of a given test suite.

Although being able to help developers to locate faulty statement, Tarantula has some limitations: I). *the colouring of statements might be misleading for debugging*. There is no absolute causal relation between the red colour and the faulty statements. In the experiments conducted by the authors, for certain cases there are no statement from the faulty program that is marked in the red colour range, *i.e.* false negative. For instance, a fault that resides in the initialization part of a program is exercised by both passed and failed test cases, thus it is marked as yellow-ish instead of reddish. And around 20% of the non-faulty statements are marked as reddish, *i.e.* false positive. II). *the input of the tool is NOT intuitive to obtain*. The tool requires the association of each statement with the result of each test case, in addition to the source code and test cases. The authors mentioned that in the experiments they instrumented the program and run the program under abundant test cases until almost every statement is exercised by at least 30 test cases. The details about how they instrumented the program and how they provided the test cases are not given in the paper. These implicit details impose heavy burdens on users of the tool.

The suspiciousness score function ω employed by Tarantula is defined as:

$$\omega_t(e,T) = \frac{\frac{n_f(e)}{N_f}}{\frac{n_s(e)}{N_s} + \frac{n_f(e)}{N_f}}$$

where the inputs are a program element e and a collection of program spectra T and

- N_s is the number of program spectra that are associated with a successful execution of the program;
- N_f is the number of program spectra that are associated with a failed execution of the program;
- $n_s(e)$ is a function that counts the number of program spectra that are associated with a successful execution of the program and involve element e;
- $n_f(e)$ is a function that counts the number of program spectra that are associated with a failed execution of the program and involve element e;

In the definition of ω_t, the component $\frac{n_f(e)}{N_f}$ represents the percentage of failing executions that involve a program element e. This component serves as the numerator within ω_t, and thus the bigger the value of the component, the higher the suspiciousness value. This embodies the intuition that the more frequently an element is involved in a failing execution of program, the more likely it is the faulty element. Similarly, the component $\frac{n_s(e)}{N_s}$ represents the percentage of successful executions that involve a program element e. The component serves as part of the denominator of $\omega_t(e,T)$, which embodies the intuition that the more frequently an element is involved in successful execution of program, the less likely it is the faulty element.

But there are some exceptions to the above intuitions. For instance, the initialization part of a program might appear in all program spectra including the ones with success execution of program. At the same time, it might be faulty itself. In this case, its suspiciousness score $\omega_t(e,T)$ would be 0.5, based on the above formula, and the faulty statements would be marked in yellow instead of red.

Overall, the suspiciousness score function of Tarantula is simple yet reasonable in a certain sense. It explains the effectiveness of the solution as well as its limitations.

Ochiai. Abreu *et al.* [5] applied the Ochiai coefficient [95] that was first used to compute genetic similarity in molecular biology as the suspiciousness score function in spectrum-based fault localization. They conducted experiments on the *Siemens fault benchmark suite* [60], and observed on average a 5% improvement on accuracy comparing to the aforementioned Tarantula [64], with an improvement of up to 30% in specific cases.

The suspiciousness score function based on the Ochiai coefficient can be represented as follows:

$$\omega_o(e,T) = \frac{n_f(e)}{\sqrt{N_f * (n_f(e) + n_s(e))}} = \frac{n_f(e)}{\sqrt{N_f * n(e)}}$$

where the inputs are a program element e and a collection of program spectra T and

- $n_f(e)$ is a function that counts the number of program spectra that are associated with a failed execution of the program and involve element e;
- $n_s(e)$ is a function that counts the number of program spectra that are associated with a successful execution of the program and involve element e;
- $n(e)$ is a function that counts the number of program spectra that are associated with any execution (either successful or failed) of the program and involve element e. We can know that $n(e) = n_f(e) + n_s(e)$, from the above definitions;
- N_f is the number of program spectra that are associated with a failed execution of the program;

To explain the intuition behind the definition of ω_o, we transform ω_o into $\omega'_o = \omega_o^2 = \frac{n_f(e)*n_f(e)}{N_f*n(e)} = \frac{n_f(e)}{N_f} * \frac{n_f(e)}{n(e)}$. The new function ω'_o retains the monotonicity property of the original function ω_o, *i.e.* $\omega_o(e_1,T) > \omega_o(e_2,T) \Longleftrightarrow \omega'_o(e_2,T) > \omega'_o(e_2,T)$, since $\forall e, \omega_o(e,T) >= 0$. Therefore, we can claim that the new function is equivalent to the original one, given the task of finding the most suspicious program element that is faulty.

The new function ω'_o consists of a product of two components: $\frac{n_f(e)}{N_f}$ and $\frac{n_f(e)}{n(e)}$. The component $\frac{n_f(e)}{N_f}$ represents the percentage of failing executions that involve a program element e, with regards to the total number of failed executions. Thus the bigger the value of the component, the higher the overall suspiciousness value. This embodies the intuition that the more frequently an element is involved in a failing execution of program, the more likely it is the faulty element. The component $\frac{n_f(e)}{n(e)}$ represents the percentage of executions that involve the element e and result in failure, with regards to the total number of appearances of the element e. This embodies the intuition that the more determined that the association between the program element e and the failed executions is, the more likely the element is the faulty element. In the extreme case, all the executions that are associated with the program element e are failed, *i.e.* $\frac{n_f(e)}{n(e)} = 1$.

However, there is a general limitation to the ω functions is that, it does not help to identify the faulty element, if all the program elements participate in the test cases at an equal frequency. For example, if all the elements are involved in all the test cases, then they will all have the same suspiciousness score. The fault could be some logic mistake in the program that fails to handle certain corner cases.

Other Statistical Techniques. Spectrum-based fault localization can be classified as a statistical technique, since its goal is to rank the most likely faulty program elements based on the historical execution information of the program. There have been a few other works [61, 85, 86, 88] that are related to this perspective. For instance, Jeffrey *et al.* [61] proposed to rank the more *suspicious* statements that could alter the result of execution, instead of considering all statements as suspicious as done for Tarantula [64] and Ochiai [5]. To find the more *suspicious* statements, the authors first instrument the program through the Valgrind [4] framework. Then they alter the output value of each statement to see whether the new output value can correct the result of failing execution, if so, the statement is suspicious to be faulty. Their experiments show that their approach is effective and outperforms Tarantula, based on the same testing suite (Siemens [60]) as used for Tarantula.

2.2.2 Program Slicing

Program slicing is a source-to-source transformation of a program, which was first introduced by Weiser [130] in 1981. Slicing is inspired by the observation that in daily work programmers perceive the source code as some specific portions of interest instead of paying attention to every detail. Over the years, the techniques of program slicing have been applied in multiple domains, such as program debugging and program maintenance. For instance, given an erroneous behaviour of a program, a programmer may examine the source code and narrow it down to the faulty code that leads to the error. The process of narrowing [136] can be achieved by program slicing. In the context of program maintenance, before making any change to the source code, a programmer needs to evaluate the impact of the changed code. To better see the impact, it is helpful to have the slices that involved the changed code.

The work of this thesis is related to program slicing in two ways. First, generally program slicing is applied to the source code of a program, whereas, we apply program slicing to assembly code, to find a closely-knit group of instructions surrounding the crashing instruction. Secondly, the goal of this thesis is to help developers debug the crash reports from the Linux kernel. We do this by identifying the offending line that triggered the crash report. Starting from the offending line, one can conduct program slicing to infer the statements that lead to the offending line, which can further help developers to find the root cause of the crash. In the rest of this section, we explain some basic concepts involved in program slicing, the general techniques to do program slicing, and the development of program slicing techniques.

Definition 2.8 (*Slicing Criterion*). A *slicing criterion* defines a specific point of interest in a program, which is the starting point from which to do slicing. A slicing criterion of a program P can be represented as a tuple $\langle s, V \rangle$, where s is a statement in P and V is a set of the variables that are involved in the statement s.

Definition 2.9 (*Backward Program Slice*). A *backward program slice* is a list of program statements that have an impact on a given slicing criterion.

Definition 2.10 (*Forward Program Slice*). A *forward program slice* is a list of program statements that are affected by a given slicing criterion.

In general, a slice is a subset of a program that preserves a specified projection of the behaviours of a program. There can be many different slices for a given program and slicing criterion. But there is always at least one slice for a given slicing criterion — the program itself.

The problem of finding a minimal slice is in general unsolvable. A simple reason for this is that in order to identify the statements involved in a slice, one should be able to determine whether a program halts, which is undecidable. Weiser [130] proved that, *there does not exist an algorithm to find statement-minimal slices for arbitrary programs*.

Although there does not exist an optimal slicing algorithm, it is possible to construct a *sub-optimal* slicing algorithm. In 1984, Ottenstein *et al.* [101] are the first ones who proposed to use PDG (Program Dependency Graph) to do slicing. A PDG is a graph that represents both the data flow and the control flow relations within the program. However, the original PDG-based slicing algorithm might yield incorrect results, since it could not handle correctly the unstructured control statements such as `goto`, `continue` and `break`, etc. Later on, slicing techniques [8, 19, 31], based on variants of the PDG, have been proposed to deal with the problem of unstructured control flow. As an alternative to PDG, in 1985, Bergeretti and Carré [27] used the notion of *information-flow relation* to construct a slicing algorithm. The information flow relation represents the value dependency relation between two variables with regarding to a statement. For instance, for the assignment statement $v_1 = v_2 + 1$, one can see that the value of the variable v_1 on exiting the statement, depends on the value of variable v_2 on entering the statement. Therefore, we say that there is an information flow relation between v_1 and v_2 regarding the statement $v_1 = v_2 + 1$. For more work about program slicing, one can refer to the survey conducted by Tip [120] .

Next we give a brief classification of the techniques of program slicing:

Static *VS.* Dynamic. The techniques of program slicing can be classified as *static* or *dynamic* slicing. The original program slicing [130] only looks at the source code to find the slices for all possible executions of program, and is later referred as static slicing. Dynamic slicing [10, 134, 135] bounds the slicing process to a certain execution history of the program. The number of slices resulting from static slicing increases exponentially with the number of branches along the slice. The result of dynamic slicing tends to be more focused, but it requires additional inputs beyond the source code, such as error-inducing test cases.

Backward *VS.* Forward. One can classify the slicing as *backward* slicing or *forward* slicing, based on the direction of slicing. Originally, program slicing [130] started from the slicing criterion

and then traced backward to search for statements that could impact the values in the slice criterion. Therefore, program slicing by default is also referred as backward program slicing. On the other hand, forward program slicing [27] looks for statements that might be affected by the values involved in the slicing criterion. Although the goals are different, backward and forward slices can be computed in a similar way.

Executable *VS.* Closure. When program slicing [130] was first introduced, the author insisted on that the resulting slice should be an executable program. The later developments of program slicing [56, 110] lose this restriction, *i.e.* the slice is not necessarily executable but a closure that respects the dependencies imposed by the slicing criterion.

2.3 Sequence Alignment

A sequence is a general concept that represents a list of elements. For instance, in `C` programs, a string is represented as a *sequence* of characters. In biology, one has *sequences* of DNA, RNA, amino acids, etc. In this section, we review the concepts and algorithms of *sequence alignment* which is applied in this thesis. Sequence alignment is an operation to align/match two sequences in order to identify the similar and dissimilar parts. Sequence alignment is frequently performed in computational biology. Its usefulness is predicated on the assumption that a high degree of similarity between two sequences often implies similar function and/or three-dimensional structure, as shown in the following scenario:

A usage scenario of sequence alignment in biology [121]. *When a new genome is sequenced, the usual first analysis performed is to identify the genes and hypothesize their functions. Hypothesizing their functions is most often done using sequence similarity algorithms, as follows. One first translates the coding regions into their corresponding amino acid sequences. One then searches for similar sequences in a protein database that contains sequenced proteins (from related organisms) and their functions. Close matches allow one to make strong conjectures about the function of each matched gene. In a similar way, sequence similarity can be used to predict the three-dimensional structure of a newly sequenced protein, given a database of known protein sequences and their structures.*

Apart from biology, the study and application of sequence alignment is also of interest in software engineering. Previous works have applied sequence alignment in software debugging [51, 90], in the study of software artefacts [107, 108], etc. We have applied sequence alignment to pinpoint the offending line of a kernel oops [47].

In the rest of the section, we first give some formal definitions about sequence alignment, and then we present some algorithms to conduct sequence alignment. Finally, we review how sequence alignment is applied in different domains, particularly software engineering.

2.3.1 Definitions

In this section, we give some formal definitions [121] about sequences, and sequence alignments.

Definition 2.11 (*Sequence*). A sequence S is represented as $S[1...n]$ where $S[i]$ represents the element at the index i within the sequence S. The sequence starts from the element $S[1]$ and ends at the element $S[n]$. The size of the sequence is n, denoted as $|S| = n$. Let Φ be the domain of all valid sequence elements. The bottom element $\perp \notin \Phi$ is referred to as a *gap*, which serves as a placeholder in a sequence. A sequence that contains solely gaps is NOT empty, *i.e.* $|S| \neq 0$.

Definition 2.12 (*Substring VS. Subsequence*). A *substring* $\widehat{S}$ is a contiguous list of elements extracted from a sequence S, denoted as $\widehat{S} = S[i...j]$ where $1 \leq i \leq j \leq n$. A *subsequence* $\tilde{S}$ is derived from a sequence S by deleting some elements without changing the order of the remaining elements.

We emphasize the difference between *substring* and *subsequence*, since they might be confused in different scenarios. In the following sections, we mainly use *substrings*.

Definition 2.13 (*Prefix VS. Postfix*). Given a *substring* $\widehat{S} = S[i...j]$ extracted from a sequence S, if $i = 1$, we also refer to the substring $\widehat{S} = S[1...j]$ as a *prefix*; if $j = |S|$, we also refer to the substring $\widehat{S} = S[i...|S|]$ as a *postfix*.

Definition 2.14 (*Global Sequence Alignment*). Given two sequences S_1 and S_2, the global sequence alignment between S_1 and S_2, can be defined as a function $G(S_1, S_2) = (S'_1, S'_2)$ where: I). S'_1 and S'_2 are sequences that may contain *gaps*. II). The sizes of sequences S'_1 and S'_2 are the same, *i.e.* $|S'_1| = |S'_2|$ III). Removing gaps from S'_1 and S'_2 leaves S_1 and S_2 respectively.

Definition 2.15 (*Local Sequence Alignment*). Given two sequences S_1 and S_2, the local sequence alignment between S_1 and S_2, can be defined as a function $L(S_1, S_2) = (S'_1, S'_2)$ where: I). S'_1 and S'_2 are sequences that may contain *gaps*. II). The sizes of sequences S'_1 and S'_2 are the same, *i.e.* $|S'_1| = |S'_2|$ III). Removing gaps from S'_1 and S'_2 leaves two **substrings** $\widehat{S_1}$ and $\widehat{S_2}$, respectively.

The definitions of global sequence alignment and local sequence alignment differ in the condition III. Given two sequences with limited size, the number of both global sequence alignments and local sequence alignments is infinite, due to the possibility of adding arbitrarily many gaps within the sequences. A local sequence alignment between two sequences can be obtained by doing a global sequence alignment between a pair of substrings extracted from the given sequences respectively. In the above usage scenario in biology, local sequence alignment is applied.

Definition 2.16 (*Element Similarity*). Let S_1 and S_2 be two sequences, one can measure the similarity of two elements $S_1[i]$ and $S_2[j]$ with a similarity function $\omega(S_1[i], S_2[j]) \in \mathbb{R}$. One can also compare the similarity between an element ϵ and a gap (the bottom element $\perp$), *i.e.* $\omega(\epsilon, \perp)$.

There are infinite number of similarity functions. In principle, the results of a similarity function can be classified into three cases: *match*, *mismatch* and *gap match*, as follows:

Match is the case where two elements are equal and neither is a gap.
$\Longrightarrow \omega(\epsilon_1,\epsilon_2)$ where $(\epsilon_1 = \epsilon_2) \wedge (\epsilon_1 \neq \perp)$

Mismatch is the case where two elements are NOT equal, and neither of them is gap.
$\Longrightarrow \omega(\epsilon_1,\epsilon_2)$ where $(\epsilon_1 \neq \epsilon_2) \wedge (\epsilon_1 \neq \perp) \wedge (\epsilon_2 \neq \perp)$

Gap Match is the case where at least one of the two elements is gap.
$\Longrightarrow \omega(\epsilon_1,\epsilon_2)$ where $(\epsilon_1 \neq \epsilon_2) \wedge ((\epsilon_1 = \perp) \vee (\epsilon_2 = \perp))$

Based on the three cases, the choices for the value assignment of each case is also infinite. But in principle, the overall assignment should be *reasonable* and well *balanced*. For instance, the similarity value of a *match* should be greater than that of a *mismatch*, and the similarity value should be proportional to the actual degree of similarity between elements in some specific domain. Here is one example of a similarity function:

$$\omega(\epsilon_1,\epsilon_2) = \begin{cases} 2, & \textbf{if } \omega(\epsilon_1,\epsilon_2) \textit{ is a match} \\ -1, & \textbf{if } \omega(\epsilon_1,\epsilon_2) \textit{ is a mismatch} \\ -1, & \textbf{if } \omega(\epsilon_1,\epsilon_2) \textit{ is a gap match} \end{cases}$$

Definition 2.17 (*Sequence Similarity*). Let S_1 and S_2 be two sequences and $|S_1| = |S_2|$. One can measure the similarity of the two sequences with a function $\Omega(S_1,S_2)$.

A simple sequence similarity function can be built on top of the element similarity function, by accumulating the similarity value for each corresponding pair of elements between the two sequences, *i.e.* $\Omega(S_1,S_2) = \sum_{i=0}^{|S_1|} \omega(S_1[i],S_2[i])$ as shown in Figure 2.3. This is a commonly used strategy to construct a similarity function.

A	C	G	**U**	$\perp$	**C**	$\perp$	$\perp$	**G**
|	|	|	|	|	|	|	|	|
$\perp$	$\perp$	U	**U**	C	**C**	A	A	**G**

Figure 2.3: The layout of two aligned sequences, where the top sequence is $S'_1 = ACGU\perp C\perp\perp G$ and the bottom sequence is $S'_2 = \perp\perp UUCCAAG$, and $\Omega(S'_1,S'_2) = -1-1-1+2-1+2-1-1+2 = 0$

In the above summation-based sequence similarity function, we compare each pair of elements *independently* without taking into account their positions in the sequences. Given the same set of element pairs, one would have the same similarity value for any order, based on the above function. In certain cases, one also takes the ordering of corresponding pairs into account, for similarity measurement. For instance, two aligned sequences with contiguous gap matches might be considered be more *similar* than the ones with evenly distributed gap matches. One then can

design another sequence similarity function that differentiates the position of aligned elements, such as $\Omega'(S_1,S_2) = \sum_{i=0}^{|S_1|} \omega'(S_1[i],S_2[i],i)$.

Definition 2.18 (*Optimal Global Sequence Alignment*). Given two sequences S_1 and S_2, and a sequence similarity function Ω, we define *optimal global sequence alignment* $OG(S_1,S_2)$ as the global sequence alignment that has the maximal value as defined by Ω, *i.e.* $OG(S_1,S_2) = (S'_1,S'_2)$, where $\forall G(S_1,S_2),\ \Omega(S'_1,S'_2) \geq \Omega(G(S_1,S_2))$.

Definition 2.19 (*Optimal Local Sequence Alignment*). Given two sequences S_1 and S_2, and a specific sequence similarity function Ω, we define *optimal local sequence alignment* $OL(S_1,S_2)$ as the local sequence alignment that has the maximal value as defined by Ω, *i.e.* $OL(S_1,S_2) = (S'_1,S'_2)$, where $\forall L(S_1,S_2),\ \Omega(S'_1,S'_2) \geq \Omega(L(S_1,S_2))$.

2.3.2 Global Sequence Alignment Algorithms

Given the above definitions, we first give the algorithms to calculate the optimal global sequence alignment. Since the algorithm to calculate the optimal local sequence alignment is quite similar with the global one, we explain it afterwards.

In order to explain the global sequence alignment algorithm, we introduce the function *Optimal Prefix Alignment Value* $V(i,j)$ below.

Definition 2.20 (*Optimal Prefix Alignment Value*). Given two sequences S_1 and S_2, and a specific sequence similarity function Ω, we define the *optimal prefix alignment value* $V(i,j) \in \mathbb{R}$ $(1 \leq i \leq |S_1|) \wedge (1 \leq j \leq |S_2|)$, as the value of the optimal global sequence alignment between the prefixes $\widehat{S_1} = S_1[1...i]$ and $\widehat{S_2} = S_2[1...j]$, *i.e.* $V(i,j) = \Omega(OG(S_1[1...i],S_2[1...j]))$.

The problem of finding the optimal global sequence alignment between two sequences S_1 and S_2, often reduces to calculating $V(|S_1|,|S_2|)$, the optimal prefix alignment value between S_1 and S_2 (a sequence is a prefix of itself). The value of V can be calculated in a *recursive* manner, following the formulas below:

$$
\begin{aligned}
(2.1)\quad & V(0,0) = 0 \\
(2.2)\quad & V(i,0) = V(i-1,0) + \omega(S_1[i],\perp) \\
(2.3)\quad & V(0,j) = V(0,j-1) + \omega(\perp,S_2[j]) \\
(2.4)\quad & V(i,j) = max(V(i-1,j-1) + \omega(S_1[i],S_2[j]), \\
& \qquad\qquad V(i-1,j) + \omega(S_1[i],\perp), \\
& \qquad\qquad V(i,j-1) + \omega(\perp,S_2[j]))
\end{aligned}
$$

1. Formula 5.1 assigns the value of aligning two empty sequences as 0. This is also the bottom rule allowing the calculation to terminate.

2. Formula 5.2 divides the optimal global sequence alignment between the prefix $S_1[1...i]$ and an empty sequence into two parts: I). aligning the tail element $S_1[i]$ with a gap $\perp$, which has the value of $\omega(S_1[i], \perp)$ II). aligning the rest of elements, *i.e.* the prefix $S_1[1..(i-1)]$, with the empty sequence, which is represented as $V(i-1, 0)$.

3. Like Formula 5.2, Formula 5.3 calculates the optimal prefix alignment value $V(0, j)$ between the prefix $S_2[1...j]$ and an empty sequence.

4. Formula 5.4 says that in order to obtain $V(i, j)$, one should take the maximal value among the three possible global sequence alignments: I). aligning the elements $S_1[i]$ with $S_2[j]$, then aligning the rest of the sequences (*i.e.* $S_1[1...(i-1)], S_2[1...(j-1)]$). II). aligning the element $S_1[i]$ with a gap $\perp$, then the rest sequences, *i.e.* $V(i-1, j)$. III). aligning the element $S_2[j]$ with a gap $\perp$, then the rest sequences, *i.e.* $V(i, j-1)$.

Based on the above formulas, one can derive a recursive algorithm to calculate the optimal global sequence alignment value, as shown in Algorithm 1.

Algorithm 1 A recursive algorithm to calculate the optimal global-sequence-alignment value

Input: sequences $S_1[1...i], S_2[1...j]$; element similarity function ω
Ouput: optimal global-sequence-alignment value $V(i, j)$

```
1:  procedure GSA
2:      if |S1| == 0 and |S2| == 0 then
3:          return 0
4:      else if |T| == 0 then
5:          return GSA(S1[1...(i-1)], S2) + ω(S1[i], ⊥)
6:      else if |S| == 0 then
7:          return GSA(S1, S2[1...(i-1)]) + ω(⊥, S2[i])
8:      else
9:          return
10:             max(GSA(S1[1...(i-1)], S2[1...(j-1)]) + ω(S1[i], S2[j]),
11:                 GSA(S1[1...(i-1)], S2[1...j])   +   ω(S1[i], ⊥),
12:                 GSA(S1[1...i], S2[1...(j-1)])   +   ω(⊥, S2[j]))
13:     end if
14: end procedure
```

The recursive algorithm is intuitive, yet inefficient. The execution of the above algorithm forms a three-way tree, where each node represents one invocation of the recursive function GSA. Therefore, its complexity is proportional to the size of the tree, which is exponential in the size of the longer sequence, *i.e.* $O(3^{max(|S_1|, |S_2|)})$.

To optimise the above algorithm, we can apply dynamic programming [25]. Indeed, the problem with Algorithm 1 is that it repeatedly recalculates intermediate results (*i.e.* $V(i, j)$ where $0 \leq i < |S_1|, 0 \leq j < |S_2|$) during the execution. If we store the intermediate results and reuse them later as shown in Algorithm 2, which is essential idea of dynamic programming, it could greatly improve the performance.

In Algorithm 2, we create a matrix dpm of size $(|S_1|+1)\times(|S_2|+1)$ to hold all the intermediate results as well as the final result. Each element $dpm[i][j]$ corresponds to an optimal prefix alignment value $V(i,j)$, *i.e.* the optimal value of the global alignment between the substrings $S_1[1...i]$ and $S_2[1...j]$.

Algorithm 2 Calculate the optimal global-sequence-alignment value by dynamic programming

```
    Input: sequences S1[1...i], S2[1...j]; element similarity function ω
    Ouput: optimal global-sequence-alignment value V(i,j)
 1: procedure GSA_dp
 2:     // creates a matrix with size (|S1|+1)×(|S2|+1)
 3:     dpm ← make_matrix(|S1|+1, |S2|+1)
 4:
 5:     // computes the Formula: V(0,0) = 0
 6:     dpm[0][0] ← 0
 7:
 8:     // computes the Formula: V(i,0) = V(i−1,0) + ω(S1[i], ⊥)
 9:     for i = 1 → |S1| do
10:         dpm[i][0] ← dpm[i−1][0] + ω(S1[i], ⊥)
11:     end for
12:
13:     // computes the Formula: V(0,j) = V(0,j−1) + ω(⊥, S2[j])
14:     for j = 1 → |S2| do
15:         dpm[0][j] ← dpm[0][j−1] + ω(⊥, S2[j])
16:     end for
17:
18:     // computes the Formula: V(0,j) = max(V(i−1,j−1) + ω(S1[i], S2[j]),
19:     //                                    V(i−1,0) + ω(S1[i], ⊥),
20:     //                                    V(0,j−1) + ω(⊥, S2[j]))
21:     for i = 1 → |S1| do
22:         for j = 1 → |S2| do
23:             dpm[i][j] ← max (
24:                 dpm[i−1][j−1] + ω(S1[i],S2[j]),
25:                 dpm[i−1][j] + ω(S1[i],⊥),
26:                 dpm[i][j−1] + ω(⊥,S2[j]))
27:         end for
28:     end for
29:
30:     return dpm[|S1|][|S2|]
31: end procedure
```

Through dynamic programming, the complexity of the Algorithm 2 reduces to be polynomial in the size of the input sequences. Overall, the entire algorithm is composed of two simple loops and one nested loop. The computation of the nest loop between the lines 21 and 28 dominates the algorithm, whose task is to populate values in the matrix dpm with $(|S_1|\times|S_2|)$ iterations. Therefore, its complexity is $O(|S_1|*|S_2|)$, where $|S_1|$ and $|S_2|$ are the sizes of the input sequences.

2.3.3 Example

We now demonstrate how the global sequence alignment Algorithm 2 works on an example. Given the same input sequences ($S_1 = ACGUCG$ and $S_2 = UUCCAAG$) and the same similarity function (ω) as in Figure 2.3, the values of the matrix *dpm* after running Algorithm 2 are shown in Figure 2.4. The algorithm fills the values one by one from top to bottom and from left to right. For each cell $dpm[i][j]$ in the matrix, the algorithm looks at its neighbouring elements in three directions (*i.e. up, left* and *up-left*), in order to derive its value $V(i,j)$. The logic corresponds to the statements between line 21 and 28 in Algorithm 2. For example, for the cell $dpm[2][4]$ (marked in a box), its neighbours in the *up*, *left* and *up-left* directions are respectively $dpm[1][4] = -1$, $dpm[2][3] = -3$ and $dpm[1][3] = -3$. To obtain the value for $dpm[2][4]$, there are three candidate alignments to evaluate. The first candidate is to align the element $S_2[2] = U$ with a bottom $\perp$ element and the preceding substring $S_2[1...1] = U$ with the substring $S_1[1...4] = ACGU$. The resulting value is $dpm[1][4] + \omega(\perp, S_2[2]) = -1 - 1 = -2$. The remaining two candidates are from the *left* direction $dpm[2][3] + \omega(S_1[4], \perp) = -3 - 1 = -4$ and the *up-left* direction $dpm[1][3] + \omega(S_1[4], S_2[2]) = -3 + 2 = -1$. As a result, the optimal value for $dpm[2][4]$ is -1, which results from aligning the elements $S_1[4]$ and $S_2[2]$, and the substrings $S_1[1...3]$ and $S_2[1...1]$. Finally, at the end of the Algorithm 2, we get the optimal value of the global alignment between the two sequences $S_1[1...6]$ and $S_2[1...7]$, which is stored at the bottom-right cell $dpm[7][6]$.

Specially, the cells that lie at the *upper* and *left* edges of the matrix have only one *relevant* neighbour to evaluate, except for the starting point $dpm[0][0]$ whose value is predefined (*i.e.* 0). The values at the upper edge (the first column $dpm[*][0]$) are the results of aligning each prefix $\widehat{S_1}$ with a sequence of gaps, and likewise the values at the left edge (the first row $dpm[0][*]$) are the results of aligning each prefix $\widehat{S_2}$ with a sequence of gaps.

	i	0	1	2	3	4	5	6
j			A	C	G	U	C	G
0		**0**	**−1**	**−2**	−3	−4	−5	−6
1	U	−1	−1	−2	**−3**	−1	−2	−3
2	U	−2	−2	−2	−3	**−1**	−2	−3
3	C	−3	−3	0	−1	**−2**	1	0
4	C	−4	−4	−1	−1	−2	**0**	0
5	A	−5	−2	−2	−2	−2	**−1**	−1
6	A	−6	−3	−3	−3	−3	**−2**	−2
7	G	−7	−4	−4	−1	−2	−3	**0**

Figure 2.4: The layout of the matrix *dpm* in the dynamic programming Algorithm 2, when the input are $S_1 = ACGUCG$ and $S_2 = UUCCAAG$

From the populated matrix *dpm*, we can extract the optimal global sequence alignments, in addition to the optimal value. Starting from the optimal value at the bottom-right, we trace backwards towards the starting point $dpm[0][0]$. Along the way, we select the neighbour that helps to yield the value of the current cell, as the next step in the trace. For instance, at the cell

dpm[2][4], we can infer that it must be the neighbouring cell dpm[1][3] that actually contributes to dpm[2][4], based on the previous formulas. In certain cases, two or three neighbours might yield the same value for the current cell, *e.g.* dpm[5][5], dpm[4][4] and dpm[3][1], *etc*. We then can choose any of them as the next step. This implies that there might be more than one alignment that yield the same optimal value. The example given in Figure 2.3 is actually one of the possible optimal global alignments, with the resulting sequences as $S_1' = ACGU\perp C\perp\perp G$ and $S_2' = \perp\perp UUCCAAG$. We mark the trace to obtain this optimal global alignment in both **bold** and red in Figure 2.4.

2.3.4 Local Sequence Alignment Algorithm

In this section, we illustrate how to obtain an optimal local sequence alignment, starting from a definition as follows:

Definition 2.21 (*Optimal Postfix Alignment Value*). Given two sequences S_1 and S_2, and a specific sequence similarity function Ω, we define the *optimal postfix alignment value* $\widetilde{V}(i,j) \in \mathbb{R}$ $(1 \le i \le |S_1|) \wedge (1 \le j \le |S_2|)$, as the value of the optimal global sequence alignment among all the postfixes that are derived from the substrings $\widehat{S_1} = S_1[1...i]$ and $\widehat{S_2} = S_2[1...j]$.

Similarly to the problem of finding the optimal global sequence alignment, the problem of finding the optimal *local* sequence alignment between two sequences S_1 and S_2 reduces to calculating the optimal postfix alignment value for all postfix pairs that are derived from the prefix pairs of the input sequences S_1 and S_2.

As a reminder, the goal of optimal local sequence alignment is to find the best pair of *substrings* that give the maximal similarity value (see Definition 2.19). To enumerate all possible substring pairs between two sequences S_1, S_2, we first enumerate all prefix pairs between S_1 and S_2, and then for each prefix pair we enumerate all the postfix pairs. Therefore, the optimal local sequence alignment value is the maximal value among all the optimal postfix alignment values $max(\widetilde{V}(i,j))$ where $(1 \le i \le |S_1|) \wedge (1 \le j \le |S_2|)$. The value of $\widetilde{V}$ can be calculated in a *recursive* manner, following the formulas below:

$$\omega(x,\perp) < 0 \quad \omega(\perp,x) < 0 \quad \omega(x,x) > 0 \quad x \neq \perp \tag{2.5}$$

$$\widetilde{V}(i,0) = 0 \tag{2.6}$$

$$\widetilde{V}(0,j) = 0 \tag{2.7}$$

$$\begin{aligned} \widetilde{V}(i,j) = max(\,0,\; & \\ & \widetilde{V}(i-1,j-1) + \omega(S_1[i],S_2[j]), \\ & \widetilde{V}(i-1,j) + \omega(S_1[i],\perp), \\ & \widetilde{V}(i,j-1) + \omega(\perp,S_2[j])) \end{aligned} \tag{2.8}$$

1. Formula 5.5 defines some constraints on the similarity function ω to make sure that the function is *reasonable*: I). aligning a non-bottom element with a bottom element has a negative similarity value, since they are dissimilar. II). aligning two identical non-bottom elements has a positive similarity value.

2. Formulas 5.6 and 5.7 state that the optimal postfix alignment value between an empty sequence and any other sequence is zero, where the resulting alignment is empty. All other alignments, which involve some gap matches, have a negative value, based on the Formula 5.5.

3. Formula 5.8 says that in order to obtain $\widetilde{V}(i,j)$, one should take the maximal value among the four possible local sequence alignments: I). an empty alignment, *i.e.* aligning nothing from the sequences. II). aligning the elements $S_1[i]$ with $S_2[j]$, then aligning the rest of the sequences (*i.e.* $S_1[1...(i-1)], S_2[1...(j-1)]$). III). aligning the element $S_1[i]$ with a gap $\perp$, then the rest of the sequences, *i.e.* $\widetilde{V}(i-1,j)$. IV). aligning the element $S_2[j]$ with a gap $\perp$, then the rest of the sequences, *i.e.* $\widetilde{V}(i,j-1)$.

Given the above rules, we can calculate $\widetilde{V}(i,j)$ recursively for a given input, starting from $\widetilde{V}(0,0)$. We give an example below in Figure 2.5 to show how it works on the same input as in Figure 2.4.

	i	0	1	2	3	4	5	6
j			A	C	G	U	C	G
0		0	0	0	0	0	0	0
1	U	0	0	0	**0**	2	1	0
2	U	0	0	0	0	**2**	1	0
3	C	0	0	2	1	1	**4**	3
4	C	0	0	2	1	0	3	3
5	A	0	2	1	1	0	2	2
6	A	0	2	1	0	0	1	1
7	G	0	1	1	3	2	1	3

Figure 2.5: The matrix of $\widetilde{V}(i,j)$, when the input sequences are $S_1 = ACGUCG$ and $S_2 = UUCCAAG$

Figure 2.5 shows a matrix of values, referred as M, where each cell $M[i][j]$ represents an optimal postfix alignment value $\widetilde{V}(i,j)$, with the input sequences $S_1 = ACGUCG$, $S_2 = UUCCAAG$, and the same similarity function ω as used in Figure 2.4 (*i.e.* ω*(match)*$= 2$; ω*(mismatch)*$= -1$; ω*(gap_match)*$= -1$). The optimal local sequence alignment value between S_1 and S_2 is the maximal value within the matrix M, which in the example is located in $M[3][5]$ (marked in a box). The position of the optimal local sequence alignment value is unknown before the calculation, which is different from the optimal global sequence alignment value that is always located in the bottom-right corner of the matrix.

Starting from $M[3][5]$ and following the trace (marked in **bold** and red), we can reconstruct the optimal local sequence alignment by re-applying the Formulas 5.6 ~5.8. The result is $S'_1 = UC$, $S'_2 = UC$, which is different from the optimal global sequence alignment (*i.e.* $S'_1 = ACGU\perp C\perp\perp G$; $S'_2 = \perp\perp UUCCAAG$). The optimal local sequence alignment usually can obtain a higher similarity value (4 *VS.* 0), compared to its optimal global sequence alignment, since it is not obliged to align the entire two sequences, where the dissimilar parts between them brings some penalty to the similarity value.

Other than the optimal local sequence alignment, one can also use the matrix M to find a list of sub-optimal local sequence alignments, which correspond to highly similar parts of the two sequences. For instance, starting from the second largest value in the matrix M, *i.e.* 3, we find a list of sub-optimal local sequence alignments, such as $S'_1 = ACG$; $S'_2 = AAG$ that are derived from the cell $M[7][3] = 3$, and $S'_1 = UCG$; $S'_2 = UCC$ starting from $M[4][6] = 3$. This insight is applied in the usage scenario of sequence alignment in biology, as we mentioned in the introduction of sequence alignment.

2.3.5 Applications of Sequence Alignments

We now discuss the origin and the applications about the concept of sequence alignment. Global sequence alignment was first proposed in 1970 by Needleman & Wunsch [100] in biology, in order to search for similarities in the amino acid sequence of two proteins. From the similarities of amino acid sequences, it is then possible to determine whether significant homology of function exists between the proteins. The homology implies the possible evolution of proteins. Later in 1981, Smith & Waterman [39] introduced the problem of local sequence alignment in biology. Both alignments are implemented based on the systematic study of dynamic programming by Belleman [25] in 1957. Finally, a number of authors have studied the question of how to construct a good scoring function for sequence comparison, to meet the matching criteria of different scenarios [11, 68].

Approximate string matching is another form of sequence alignment, which is the technique of finding strings that matches a pattern *approximately* (rather than exactly). Formally, approximate string matching can be defined as: Given a pattern string $P = p_1p_2...p_m$ and a text string $T = t_1t_2...t_n$, find a substring $T_{i,j} = t_it_{i+1}...t_j$ in T, which, of all substrings of T, has the smallest *edit distance* to the pattern P. One common instance of edit distance, called Levenshtein distance [82], is the minimal number of single-character edits (*i.e.* insertion, deletion or substitution) required to change one string into the other. Approximate string matching with Levenshtein distance as measure can reduce into *local* sequence alignment, since the deletion and insertion operations of string matching are equivalent to gap matches in sequence alignment and the substitution is equivalent to a mismatch.

By changing the metrics of edit distance, we can transform approximate string matching into several other interesting problems. For instance, if we only allow insertion and deletion, not

substitution, then we can apply approximate string matching to solve the problem of longest common subsequence (LCS) [55], which is to find the longest common subsequence of all sequences in a set of sequences (often just two). If we allow only the substitution operation, the edit distance becomes Hamming distance [50] that is the minimum number of substitutions required to change one string into the other. Hamming distance was initially used to model the bit flipping error in telecommunication. It was later applied in several disciplines including information theory, code theory and cryptography. As another instance of edit distance, Damerau-Levenshtein distance [34] extends Levenshtein distance, to allow a transposition of two adjacent characters (*i.e.* switching positions), in addition to insertion, deletion and substitution. Since the above four operations correspond to more than 80% of human misspellings, as observed by Damerau [34], Demarau-Levenshtein distance is used to detect and correct spelling errors.

Sequence alignment has also been applied in software engineering, especially on mining software repositories (*i.e.* collection of data about software artifacts, such as source code, or bug reports, *etc.*). Global sequence alignment itself presents a method to measure the similarity between two sequences, which can serve as a similarity function during the data clustering. Han *et al.* [51] applied sequence alignment on call traces (*i.e.* sequences of function calls) to cluster error reports that might originate from certain performance related bugs. Lo *et al.* [90] mined program execution traces, which are sequences of state transitions generated from the Finite State Automata that are derived from the source code. They clustered the program execution traces in order to infer API specifications automatically. Ramanathan *et al.* [107] studied another form of program execution traces that are logs generated from the execution of instrumented binaries. They aimed to identify the longest common sequence between traces generated from two revisions of program, to allow developers to perform impact analysis more accurately during program testing and software maintenance.

2.4 Summary

In this chapter, we have covered some background knowledge about this thesis. The main goal of this thesis is to help Linux kernel developers better debug the faults that are encountered by users. Therefore, overall, our work falls into the domain of ***software debugging***. Particularly, we focus on the scenario of *error report debugging*, which, in the case of Linux kernel debugging, is more common and practical than the *bug reproduction* scenario. We have given some detailed reviews on each of the above subjects.

In this thesis, we proposed a technique to pinpoint the offending line from a kernel oops, which gives Linux kernel developers an important clue to proceed oops debugging. Thus, our work can be further classified as one of the techniques of ***software fault localization*** which is one of the most critical activities that are involved in software debugging. We then discussed some state-of-the-art techniques in this domain, including *spectrum-based fault localization* and

program slicing.

The technique we have designed to pinpoint the offending line is inspired by the sequence alignment algorithms which were initially proposed in bioinformatics domain. To give more insight about sequence alignment algorithms, we give some background knowledge about ***sequence alignment*** in this chapter. Particularly, we presented the definitions and the algorithms for the *global* and *local* sequence alignments, and their applications in other domains.

CHAPTER 3

ALL ABOUT KERNEL OOPS

Kernel oopses collected from real world deployments possess knowledge that is of interest to Linux kernel developers. If extracted, the knowledge can help developers better understand the issues that trigger kernel oopses. To this end, in this chapter, we perform a study of over 187,000 Linux kernel oopses, collected in the oops repository (`oops.kernel.org`) maintained by Red Hat between September 2012 and April 2013. We consider the overall features of the data, the degree to which the data reflects other information about Linux, and the interpretation of features that may be relevant to reliability. We find that the data correlates well with other information about Linux, but that it suffers from duplicate and missing information. We furthermore identify some potential pitfalls in studying features such as the sources of common faults and common failing applications.

The rest of this chapter is organized as follows. Section 3.1 illustrates the key features of a kernel oops, and the workflow around the generation and transmission of a kernel oops. Section 3.2 gives an overview of the data and studies its internal consistency, focusing on the possibility of duplicate or missing oopses. Section 3.3 compares properties of the oopses to existing information about the Linux kernel, including versions and their release date, distributions and their associated versions, hardware architectures, and the most error-prone services. Then, Section 3.4 studies some features of the data that can be relevant to understanding kernel reliability, including the frequency of different error types, failing functions, reasons for entering the kernel, and the impact of previous kernel events. Section 3.5 then describes the threats to the validity of our study. Finally, we conclude in Section 3.6.

3.1 Background

We first present the key features of kernel oopses, the code that triggers their generation, and the workflow that results in their being submitted to the repository maintained by Red Hat.

3.1.1 Key Features of a Kernel Oops

A kernel oops [123] consists of the information logged by the Linux kernel when a crash or warning occurs. It is composed mostly of *attribute*: *value* pairs, in which each *value* records a specific piece of information about certain attribute of the system. The format and content of a kernel oops depend somewhat on the cause of the oops and the architecture of the victim machine. We focus on the x86 architecture (both 32-bit and 64-bit), which currently represents almost all of the oopses found in the kernel oops repository.

Figure. 3.1 shows a (slightly simplified) sample kernel oops[1] from the repository. This example illustrates a typical kernel oops generated by x86 code in the case of a runtime exception. Some values are omitted (marked with ...) for brevity. We highlight the key features, and explain them as follows:

Oops description A kernel oops begins with a description of the cause of the oops. Our oops was caused by NULL-pointer dereference (line 1).

Error site The *IP* (Instruction Pointer) field indicates the name of the function being executed at the time of the oops, the binary code offset at which the oops occurred in that function, and the binary code size of that function. This information is also indicated by the value of the registers EIP for the 32-bit x86 architecture and RIP for the 64-bit x86 architecture. Lines 2, 9 and 26 each show that in our example oops, the error occurred within the function `anon_vma_link`.

Die counter The *Oops* field includes information about the die counter, between square brackets, which indicates how many serious errors have occurred since the system was booted. Our oops is the third such error (line 3).

Process name The *comm* field indicates the name of the process being executed when the error occurred. Our oops was generated during the execution of `gnome-panel` (line 13).

Taint The *Tainted* field gives a sequence of characters, amounting to a bitmap, recording whether some events have previously occurred in the kernel execution. In our example, the Tainted field on line 7 contains `'G'` indicating that no proprietary module has been loaded into the kernel and `'D'` indicating that a kernel oops has previously occurred. The latter is consistent with the value of the die counter.

[1] ID in the kernel oops repository maintained by Red Hat: 5095a67440bfca031f0007e3

```
1  BUG: unable to handle kernel NULL pointer dereference at (null)
2  IP: [<c10a1ca1>] anon_vma_link+0x24/0x2b *pde = 00000000
3  Oops: 0002 [#3] SMP
4  last sysfs file: /sys/devices/LNXSYSTM: 00/LNXSYBUS:00 /PNP0C0A:
     00/power_supply/BAT1/charge_full
5  Modules linked in: rndis_wlan rndis_host cdc_ether...
6     [last unloaded: scsi_wait_scan]
7  Pid: 2452, comm: gnome-panel Tainted: G D (2.6.32-5-6 #1) Aspire 5920
8  EIP: 0060: [<c10a1ca1>] EFLAGS: 00010246 CPU: 0
9  EIP is at anon_vma_link+0x24/0x2b
10 EAX: f6f84404 EBX: f6f84400 ECX: eb4aa5b4 EDX: 00000000
11 ESI: eb4aa580 EDI: eb4aa5d8 EBP: ef76a5d8 ESP: f61c3eb8
12 DS: 007b ES: 007b FS: 00d8 GS: 00e0 SS: 0068
13 Process gnome-panel (pid: 2452, ti=f61c2000 task=ef4a1100 task.ti=f61c2000)
14 Stack:
15   00000006 ef76a630 c102efe8 d42a7a40 00000000 00000004...
16 Call Trace:
17   [<c102efe8>] ? dup_mm+0x1d5/0x389
18   [<c102fb0c>] ? copy_process+0x91b/0xf2d
19   [<c1030258>] ? do_fork+0x13a/0x2bc
20   [<c10b1f41>] ? fd_install+0x1e/0x3c
21   [<c10b9504>] ? do_pipe_flags+0x8a/0xc8
22   [<c113c603>] ? copy_to_user+0x29/0xf8
23   [<c1001dae>] ? sys_clone+0x21/0x27
24   [<c10030fb>] ? sysenter_do_call+0x12/0x28
25 Code: 02 31 db 89 d8 5b c3 56 89 c6 53 8b 58 3c 85 db...
26 EIP: [<c10a1ca1>] anon_vma_link+0x24/0x2b SS: ESP 0068: f61c3eb8
27 CR2: 0000000000000000
28 ---[ end trace 4dbb248fc567ac92 ]---
```

Figure 3.1: A sample kernel oops

Version Following the taint bits, the *Tainted* field indicates the build version of the Linux kernel and possibly the machine model, defined by the vendor. Line 7 shows that our oops was generated by Linux version 2.6.32.

Call trace The *call trace* contains the list of return pointers into the sequence of nested function calls that led to the oops. Lines 16-24 show the call trace of our oops.

Oops ID A kernel oops may end with a string of the form `--[ end trace XXXX ]--`, where `XXXX` represents the identifier associated with the kernel oops. The identifier (*oops_id*) is represented as a 16-character hexadecimal string. It represents the current value of a global variable that is initialized to a 64-bit random number on the first oops within a given boot and is then

Table 3.1: The information found in common kinds of oopses

oops type	id	version	taint	die cnt	cmd	stack top	call trace	general cause
CNTXT_BLOCK		×	×		×		×	*blocking call in a non-blocking context*
CNTXT_SCHED		×	×		×		×	*call schedule in an atomic context*
CNTXT_CALL		×	×		×		×	*func call violates context dependent rules*
PT_MAP		×	×		×		×	*invalid entry in the page mapping table*
PT_STATE		×	×		×		×	*inconsistent or invalid page table*
BUG_ON	×	×	×	×	×	×	×	*assertion failure*
INV_PTR	×	×	×	×	×	×	×	*null pointer/bad page dereference*
WARN	×	×	×		×	×	×	*warning*
SOFT_LOCK		×	×		×	×	×	*one CPU is stuck*
GPF	×	×	×	×	×	×	×	*violation of hardware protection mechanisms*

incremented each time an oops containing an oops id is generated. Line 28 shows the oops id of our oops.

3.1.2 Types of Kernel Oopses

While many of the kernel oopses in the Red Hat repository contain the above features, this is not always the case. Indeed, a kernel oops is generated by ad-hoc print statements in the kernel code. Thus, the format of an oops can vary for each error type. Furthermore, because oopses include architecture-specific information, such as register names and memory addresses, the format of a kernel oops can vary by architecture.

Most of the text in the oops shown in Table 3.1 is generated by the function `__die`, shown in Figure. 3.2. This function is used for most severe errors. In `__die`, most of the work is done by the call to `show_registers` (line 7), which prints the registers but also prints other information, including the call trace. The function `__die`, however, is not used for all kinds of errors. For example, when the memory manager detects a page containing invalid flags, it uses `bad_page` (Figure. 3.3) to generate an oops. `bad_page` does not use `show_registers`. Instead it uses `dump_stack`, which prints the call trace, but not the register information. Finally, a further variation in the oops structure is introduced by the fact that generation of the call trace is ultimately controlled by a configuration flag. Table 3.1 lists a collection of common x86 error types and the features that may be included in the corresponding oopses.

In addition to `__die` and to ad hoc oops generating functions, such as `bad_page`, the Linux kernel also provides the oops-generating macros `BUG` and `WARN`, and variants, for generating bug and warning messages, respectively. These macros are executed for diverse reasons, but generate their oopses in a fixed format. We designate oopses generated by the `WARN` macros as warnings and all others as bugs.

```
1  int __kprobes __die(const char *str, struct pt_regs *regs, long err)
2  {
3          unsigned short ss;
4          unsigned long sp;
5          printk(KERN_EMERG "%s: %04lx [#%d] ", str, err&0xffff, ++die_counter);
6          printk("SMP \n");
7
8          if (notify_die(DIE_OOPS, str, regs, err, ...) == NOTIFY_STOP)
9                  return 1;
10         show_registers(regs);
11         ... // initialize ss, sp
12         printk(KERN_EMERG "EIP: [<%08lx>] ", regs->ip);
13         print_symbol("%s", regs->ip);
14         printk(" SS:ESP %04x:%08lx\n", ss, sp);
15         return 0;
16 }
```

Figure 3.2: Function `__die` (simplified), called on many kinds of system faults

```
1  static void bad_page(struct page *page)
2  {
3          ... // code to limit the frequency of reports
4          printk(KERN_ALERT "BUG: Bad page state in process %s pfn:%05lx\n",
5                  current->comm, page_to_pfn(page));
6          printk(KERN_ALERT
7                  "page:%p flags:%p count:%d mapcount:%d
8                   mapping:%p index:%lx\n",
9                  page, (void *)page->flags,  page_count(page),
10                 page_mapcount(page), page->mapping, page->index);
11         dump_stack();
12     out:
13         __ClearPageBuddy(page);
14         add_taint(TAINT_BAD_PAGE);
15 }
```

Figure 3.3: Function `bad_page` (simplified), called on detecting a bad page

3.1.3 Workflow around Kernel Oopses

The life cycle of a kernel oops is composed of three phases: *generation*, *collection*, and *submission*. Our understanding of this workflow is based on our study of the related tools and on our experiments in which we generated an oops and observed its path to the repository.

The Linux kernel itself is in charge of *generation*, which is ultimately performed through the kernel printing function `printk`. This function writes to the kernel message buffer (*/proc/kmsg*). The generated message is then asynchronously fetched by the kernel service *'klogd'*, which outputs the contents of the buffer to the system logging files. This strategy implies that some oops messages may be missed or corrupted, if *e.g.*, the message buffer overflows, or if the kernel

crashes before the message is picked up by *klogd*.

Collection and *submission* are then taken care of by tool sets that are bundled with various distributions of Linux. Fedora provides *ABRT* (the Automatic Bug Reporting Tool) [2], which collects `C/C++` and `Python` application crash reports, and as well as kernel oopses. *ABRT* consists of several services, including abrtd (daemon to detect crashes), abrt-gui and abrt-applet (GUI for the management of error reports), and abrt-oops (kernel oops parser). Debian and Ubuntu provide *kerneloops* [3], which is dedicated to kernel oopses. *kerneloops* includes the kerneloops daemon, kerneloops-applet and kerneloops-submit, to collect kernel oopses, to ask for permission to submit them, and to submit them, respectively. Different toolsets have different strategies for storing, parsing, and submitting kernel oopses. For example, *ABRT* truncates the end marker of a kernel oops including the *oops_id*, while *kerneloops* preserves this information.

The kernel oops toolsets submit oopses to both the public repository `oops.kernel.org` [12] and the corresponding repository for each Linux distribution. *ABRT* first parses kernel oopses from system logging files, stores them in its local file system, and then asks the user whether to submit them. It keeps the parsed kernel oopses persistent and hashes the content to detect duplicates. *Kerneloops* keeps oopses in memory after parsing, and detects duplicates only within these known oopses. *Kerneloops* thus risks submitting duplicate oopses if it is restarted. We have reproduced this behavior.

3.2 Properties of the Raw Data

We now study the data in the Red Hat kernel oops repository. We first give an overview of the sources of these kernel oopses, in terms of the associated Linux kernel version and the associated Linux distribution. We then consider the possibility of duplicate or missing oopses.

Oops origins As shown in Table 3.1, most oops kinds contain architecture-specific information, such as register names and memory addresses. We consider only architecture-specific oopses for the x86 processor family, including both 32-bit and 64-bit x86 architectures, as well as all non architecture specific oopses. This amounts to over 99% of the available 187,342 oopses.

Figure. 3.4 shows the number of reports for each Linux kernel version represented in the repository. The version information is taken from the Tainted field, as described in Section 3.1.1. The Linux kernel is made available as a series of subreleases, starting with a series of "rc" (release candidate) versions before the primary release, followed by the release itself, followed by a series of bug-fixing releases, having increasing minor release numbers (*e.g.*, 2.6.32.14 or 3.6.4). We do not distinguish between these subreleases. For 7% of the reports, there is no version information. These reports are typically associated with oopses for which the entire report is a single line of text indicating the problem.

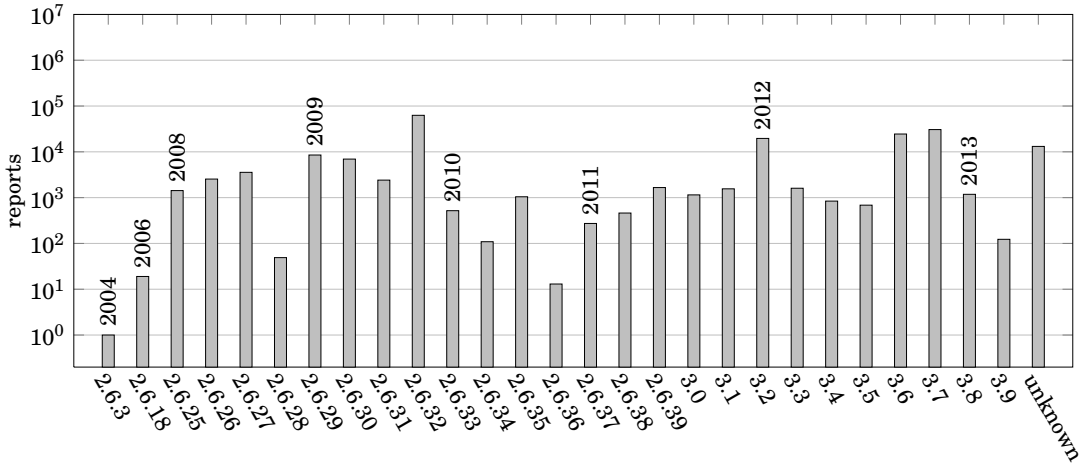

Figure 3.4: Number of reports per Linux version (log scale) and release year

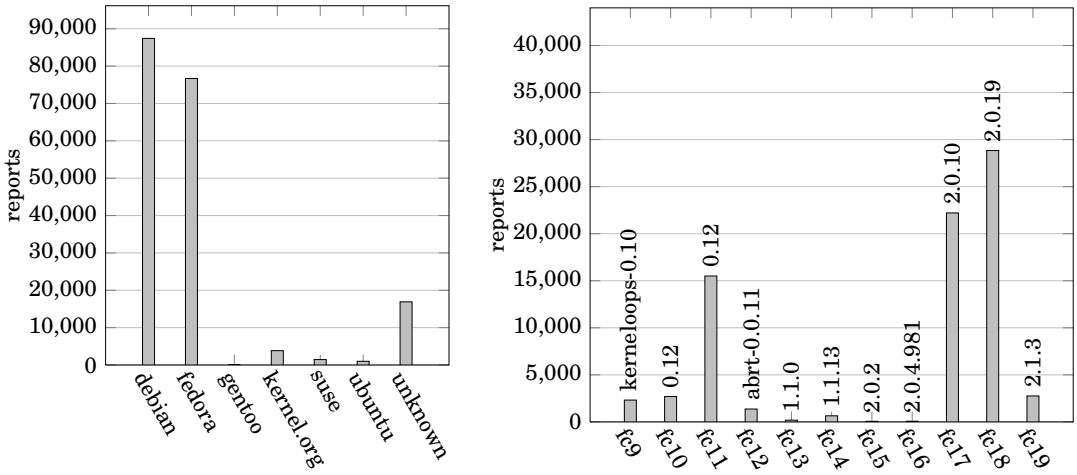

Figure 3.5: Number of reports per Linux distribution and reporting tool used

Figure. 3.5 shows the number of reports per Linux distribution, with the Fedora information broken down into the individual Fedora releases. Oopses contain no specific field for distribution information. Instead, we try to deduce the distribution from the version information. Some distributions, such as Fedora or Ubuntu, include a string indicating the distribution explicitly, with Fedora also including the number of the Fedora release. In other cases, such as Debian and Suse, we have used other information, such as filename paths or comments related to the version name found on the Internet. We find that most reports are for Debian or Fedora. This is likely due to the support in these distributions for kernel oops reporting, via the kerneloops and ABRT

utilities, rather than the popularity or reliability of the distributions themselves.

The number of reports from Fedora varies widely by release, with Fedora 11, 17 and 18 having the most reports. The number of reports seems to be mainly affected by the strategy taken by Fedora for collecting and transmitting the reports. For example, whether the oops reporting tool is installed by default, and whether it asks the user for permission to send a report or whether reports are sent automatically. Red Hat appears to have experimented with sending more and more reports automatically in Fedora 17 and 18, *i.e.*, once the new repository was available. Nevertheless, this effort seems to have been rolled back at the end of the considered period, due to an outage of the repository between late February and early March 2013.

In some cases, the version information contains only the number of the Linux version, with no added distribution information. In these cases, we consider that the kernel is not associated with a particular distribution, and instead has been specifically compiled and installed by the user from the code at the Linux kernel website, kernel.org. Such oopses may come from more advanced users.

Duplicate oopses Ideally, each error that occurs in the system would correspond to exactly one oops in the repository. In practice, however, we find many probable duplicates.

Concretely, we have observed the following kinds of probable duplicates: *i.* Identical reports. We detect these by using the SHA1 hash of each original raw message. *ii.* Reports that are identical modulo fragments of kernel time stamps or log level markers. Such text is normally removed by the parsers of the kernel oops tool sets. For duplicate detection, we remove these timestamps before computing the SHA1 hash. *iii.* Finally, a prefix or suffix of a report may be missing. For duplicate detection, we parse the reports, and consider to be duplicates reports for which the only difference as compared to another oops is that some fields are missing. We refer to such reports as being *subsumed*.

Figure. 3.6 shows the percentage of reports that may be duplicates of another oops for each Linux distribution, considering those found to be identical by the SHA1 hash and by subsumption. Overall, we observe a rate of duplicates of 40%. Further, in Figure. 3.7, we look closely at the duplicates from one particular Linux distribution, Fedora, which is one of the main contributors to the data in the repository. We observe that, although Fedora has used ABRT, having a sophisticated duplicate detection mechanism, since Fedora 12, many of the Fedora reports, particularly the warnings, appear to be duplicates. Recall, however, that ABRT discards the oops id, so we are missing a crucial piece of information that could distinguish reports. Figure. 3.8 shows the percentage of oopses with no oops id, for versions having at least 1000 reports. Finally, Debian has the largest number of duplicates, at over 10,000. However, it has the largest number of reports overall (Figure. 3.5), and so this number makes just over 12% of its total.

In summary, Linux distributions that are supposed to result in few duplicates, seem to result in many, although the oops id is missing so we cannot be sure. And Linux distributions that are supposed to be likely to result in duplicates do result in duplicates, but at a relatively low

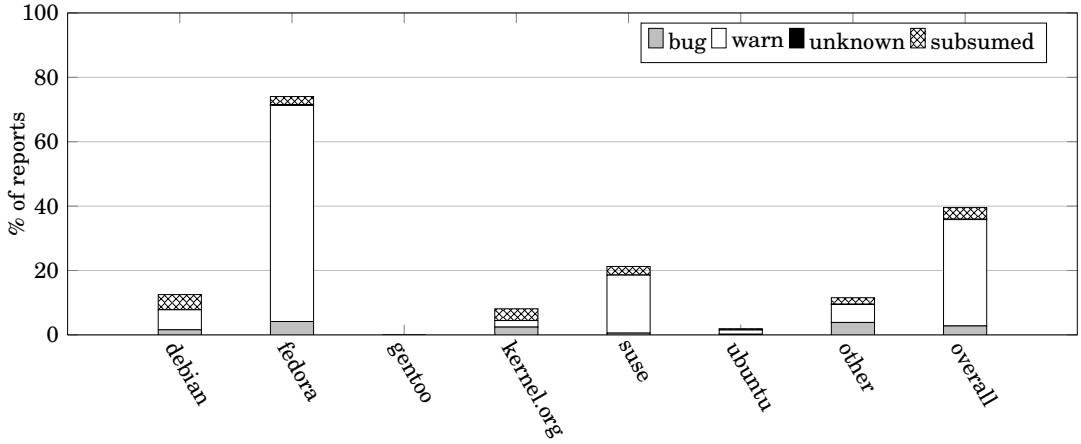

Figure 3.6: Duplicate reports per Linux distribution

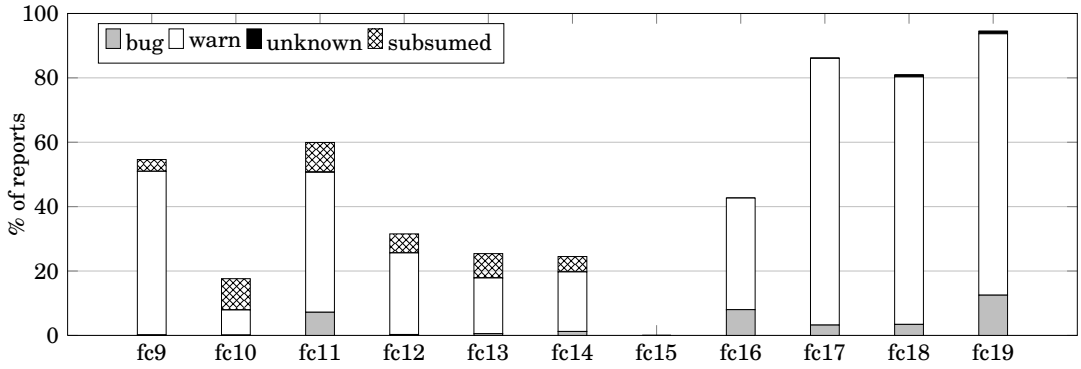

Figure 3.7: Duplicate reports per Fedora release

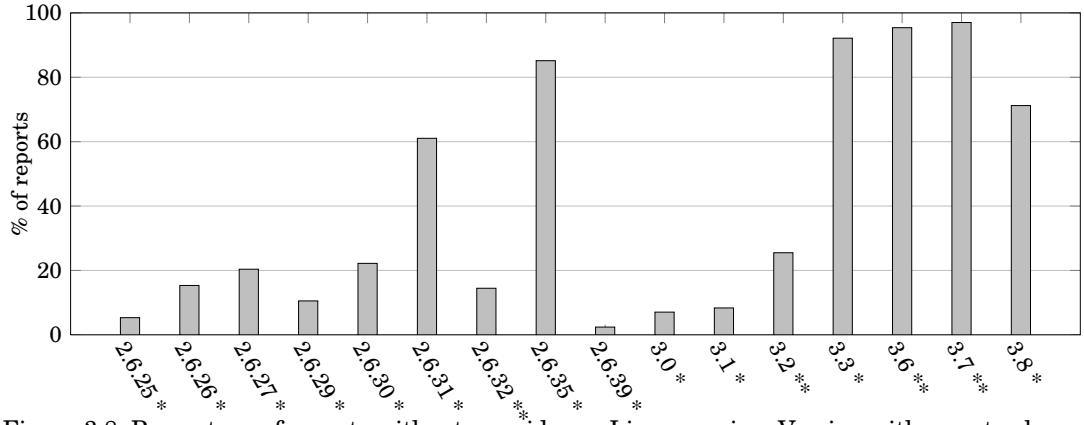

Figure 3.8: Percentage of reports without oops ids per Linux version. Version with one star have at least 1000 reports, while versions with two stars have at least 10,000 reports. This convention is followed in subsequent figures.

rate. Thus, subsequently, we consider all reports, whether or not they appear to be duplicates. This strategy may magnify the importance of some kinds of oopses, in particular warnings, which are the most common possibly duplicated reports, but it does accurately reflect the kind of information that is provided to the user of the repository.

Missing oopses It is intrinsically difficult to accurately count the number of missing reports, because such reports are not present in the repository. Nevertheless, we can estimate their number from the various counters and historical information that are found in some kinds of reports (see Table 3.1). Specifically, we consider *i.* the oops id, *ii.* the die counter, and *iii.* the taint. The oops id is a random number generated on the first oops id generating oops on each boot and incremented for each subsequent such oops. Because the initial value is random, gaps may indicate either missing reports or the starting value for a new boot. Thus, the oops id itself is not a reliable means of detecting missing reports. The die counter is initially 0, and is incremented and then printed on several kinds of oopses. When the die counter is greater than 1, there should be a preceding oops in the same boot with the preceding die counter value. Any such oops id that is not present represents a missing oops. Finally, the taint word in an oops indicates whether there is at least one previous bug or warning within the current boot. For a report with either 'D' (die) or 'W' (warn) taint and with an oops id, the immediately previous oops id should also be present. Again, any such oops id that is not present represent a missing report.

Over all of the versions, we find 7852 oopses where there is an oops id and the die counter reaches a value greater than 1, and 51% of these where the expected previous die counter is missing. Furthermore, we find 37265 oopses where there is an oops id and the taint indicates a previous bug or warning, and 47% of these where the expected previous oops id is missing. In each case, we require that the version information be present and identical, to try to avoid confusion between oops ids that are identical but unrelated. These results consistently show a high rate of missing oopses in cases where multiple oopses occur within a single boot.

While our results suggest a high rate of missing oopses, the validity of this conclusion is called into question by the small number of oopses taken into account: 4% in the die counter case and 20% in the taint case. The scope of our analysis is indeed constrained by the need to take into account the oops id. As we have noted, Fedora's ABRT, used starting with Fedora 12, removes the oops id from a kernel oops. While only 0.2% of the reports have a die counter greater than 1 but no oops id, 11% have taint indicating a previous bug or warning but no oops id. Thus, it is impossible to assess missing oopses in these cases. Nevertheless, based on the ratio of the number of oopses missing according to the taint information to the total number of oopses plus the missing number, it appears that overall, we are missing at least 9% of the oopses.

The oops id In principle, the problem of detecting duplicate and missing reports could be resolved if the kernel would associate a unique identifier with each boot, and a counter, analogous to the die counter, that is incremented on each oops. A step in this direction is already present

with the oops id. Nevertheless, some observations about the oops id reveal challenges that can be encountered in implementing such a scheme.

To be useful, the proposed boot identifier must be unique. We have observed that the current oops id is not always unique. In the extreme, the oops ids `4eaa2a86a8e2da22`, `a7919e7f17c0a727`, and `0000000000000002` occur with 35, 27, and 21 different version strings, respectively. Overall, we have 28 such non-unique oops ids. Some may be due to corruption of the oops id counter. Others may be due to incrementing such a corrupted value; for instance, we also find `0000000000000003`, etc., up to `0000000000000008`, in decreasing amounts. Still it is possible that some of these duplicated oops ids result from weaknesses in the random number generator.

Starting in Linux 3.3,[2] the implementation of the Linux random number generator was improved to take advantage of hardware-level random number generation facilities, if available. Such facilities are available for the x86 architecture. Indeed, after Linux 3.3, our only duplicates, taking into account the version string, are `0000000000000002` and nearby oops ids, as well as `a7919e7f17c0a727`. These may thus represent corrupted oops id values. On the other hand, as shown in Figure. 3.8, we also have relatively few oopses for those versions that contain oops ids.

Likewise, for the proposed counter to be useful, the process of incrementing it on each oops must be well-defined. Currently, in the Linux kernel, neither the oops id nor the die counter is incremented atomically, meaning that these computations are subject to race conditions. Indeed, introducing locks could reduce the amount of information available, *e.g.*, if the lock itself were to become corrupted due to the error.

Finally, in the context of the repository, the boot id and counter have to be transmitted to the repository reliably. This is not he case of recent Fedora oopses, where ABRT strips the oops id.

Stale pointer A kernel oops normally contains a call trace which lists the function invocation trace to the crashing function. However, correctly using the call trace to find the caller of the crashing function, could be tricky, and requires understanding the kernel call trace generation process. As an optimization, the kernel is compiled such that the stack frames do not contain a frame pointer. This optimization implies that the kernel is not able to unambiguously identify return pointers on the stack. Instead, it scans the stack to find addresses that could correspond to an address within a function. For each such address, it determines whether its position corresponds to the expected offset from its stack base pointer register. If this property is not satisfied, the pointer is referred to as *stale* and the name of the corresponding function is annotated with a question mark ? in the *call trace* field of kernel oops. *Stale pointers* are common, and may arise when function pointer arguments are passed via the stack or when the stack contains uninitialized local variables whose locations coincide with the positions of previously stored return pointers. It may also occur that the base pointer register does not correspond to any of the stack positions containing return pointers, and thus the entire call trace is considered

[2] Kernel patch cf833d0b9937874b50ef2867c4e8badfd64948ce

to be stale. The latter phenomenon is illustrated by the call trace in our sample kernel oops in Figure. 3.1 (lines 17 to 24).

To address the issue of stale pointers, when consulting the call trace, we find the service associated with the topmost non-stale pointer, if available. If there is no non-stale pointer, we fall back on taking the function at the top of the call trace.

3.3 Correlation with External Information

To help establish the representativeness of the data in the kernel oops repository, we compare the properties of the oops data to some independent observations about the Linux kernel. In this, we study the kernel versions and Linux distributions associated with the oops reports, and the architectures from which the reports originate.

Linux version. As shown in Figure. 3.4, we have at least 1000 reports from Linux kernel versions ranging from 2.6.25, released in April 2008, to 3.8, released in February 2013. Four versions, from Linux 3.6 to Linux 3.9, were released in the period covered by our study. For the recent versions, we might expect that the number of reports would increase when the version is released, then remain stable for the period in which it is the most recent version, and then finally decrease when a new release appears. Earlier versions are typically used by users who have a need for stability, rather than new functionality. We might expect the number of reports from such versions to be fairly stable, but to decrease over a long period of time.

The top two graphs of Figure. 3.9 show the number of reports per day for the older versions for which we have the most reports: 2.6.29, 2.6.30, 2.6.32, and 3.2. For these versions, the number of reports per day is fairly stable, modulo a few spikes and periods in which there are no reports at all. Excluding the days on which we have no reports, 2.6.29 has on average 39 reports per day, 2.6.30 has on average 32 reports per day, and 2.6.32 has on average 287 reports per day. In each case the number of reports per day is relatively constant across the considered time period. These results substantiate our hypothesis. However, for Linux 3.2, we have only 90 reports per day, which is fewer than for Linux 2.6.32, even though Linux 3.2 is more recent. Thus, it appears to be necessary to take into account other factors than just the age of the Linux version.

The bottom graph of Figure. 3.9 shows the number of reports per day for the recent versions for which we have the most reports: 3.6 and 3.7. As expected, the number of reports for Linux 3.6 increases sharply in early November 2012, shortly after the release of Linux 3.6 at the end of September 2012, and the number of Linux 3.6 reports starts to decline in late January 2013, as the number of reports from Linux 3.7, which was released in December 2012, rises. Note that in each case, reports start to appear in large numbers roughly one month following the release date. This can be ascribed to the fact that users typically do not adopt each version as it is released, but wait for the subsequent more stable subreleases, in which essential bug fixes have been applied.

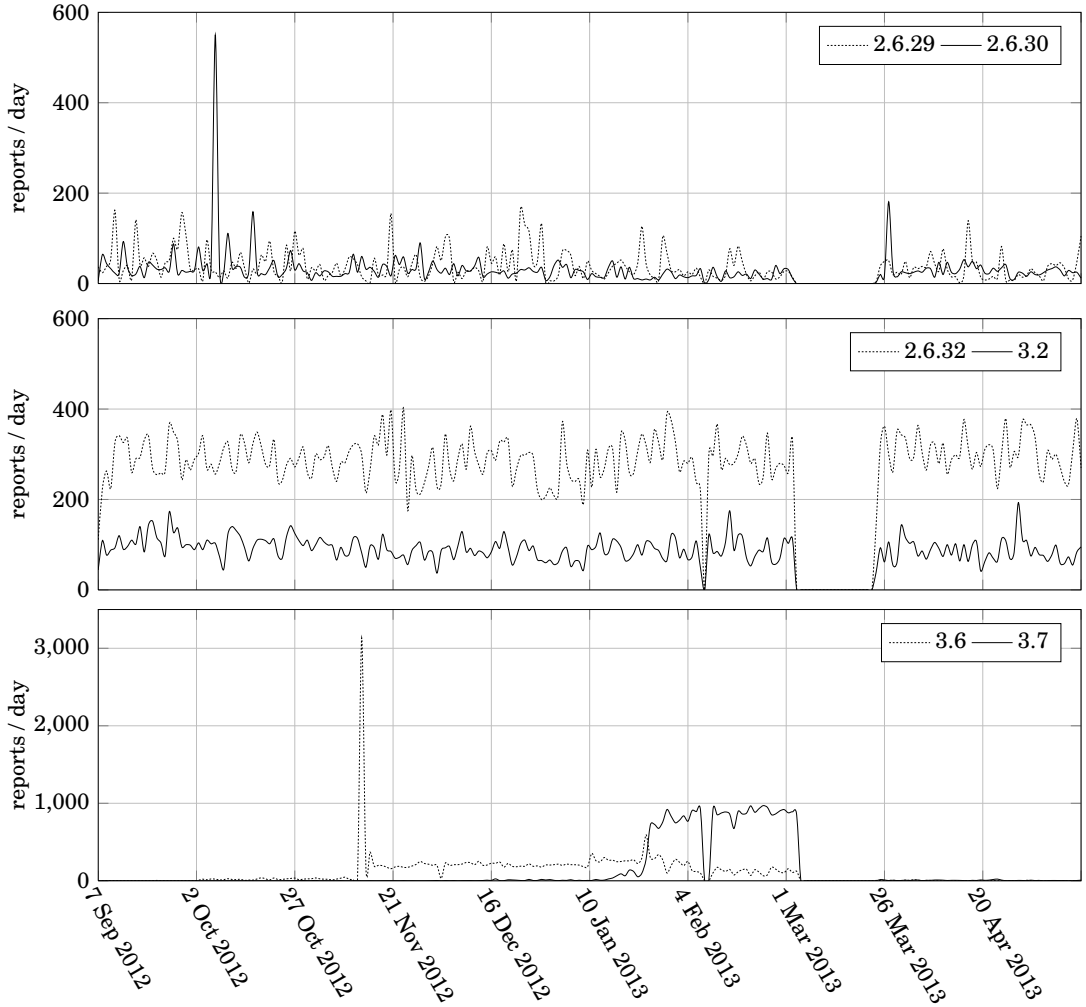

Figure 3.9: Prevalence of reports from selected Linux versions over time (versions for which there are at least 5000 reports)

While versions 3.6 and 3.7 fit the expected pattern, we do not have the same pattern with Linux 3.8, which was released in mid February 2013, well within the considered period. We have very few reports for this version, typically 25 or fewer per day. Again, we conclude that other factors must be involved.

Linux distribution. To better understand the reason for the number of reports per version, we consider also the association of the various Linux versions with the various Linux distributions (*e.g.,* Fedora, Debian, Ubuntu etc.). Indeed, most users of the Linux kernel use the kernel provided with their Linux distribution, and upgrade the kernel as the distribution suggests to do so.

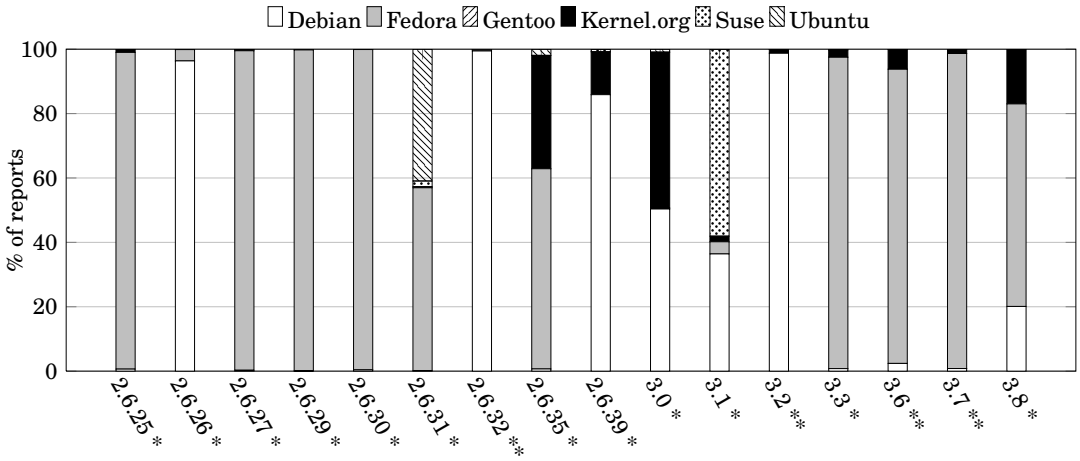

Figure 3.10: Prevalence of reports from different distributions for different versions, for which there are at least 1000 reports (* referred to Figure 3.8)

Figure. 3.10 shows the prevalence of kernel oopses for each Linux kernel and for each Linux distribution, for those versions for which we have at least 1000 reports. Most versions have most reports from either Debian or Fedora, but not both. Reports from Linux 2.6.26, 2.6.32, and 3.2 are almost entirely from Debian. Linux 2.6.26 was the kernel of Debian 5.0, released in 2009, Linux 2.6.32 was the kernel of Debian 6.0, released in 2011, and Linux 3.2 is the kernel of Debian 7.0, released in 2013. In particular, within the considered time period, Debian 6.0 was the current "stable" version, used by most Debian users, and Debian 7.0 was the current "testing" version. This explains the constant, relatively high number of reports for Linux 2.6.32, observed in Figure. 3.9, and the constant but lower number of reports for Linux 3.2. We expect that the number of reports for Linux 3.2 will rise, now that it is used in the current Debian "stable" version, as of May 2013.

As shown in Figure. 3.10, many of the versions not used by the Debian releases mostly have reports from Fedora. Figure. 3.11 breaks down all of the versions for which we have any report from Fedora by the rate of the various Fedora releases as compared to the number of Fedora reports for that version. We find that the earliest Linux version associated with each Fedora release in the oopses is always the one that was shipped with that Fedora release. For instance, oopses from Fedora 9 appear with Linux 2.6.25, which is the default kernel of Fedora 9. Furthermore, we also see how the Linux kernel shipped with a Fedora release changes over time, and then how one Fedora release is subsumed by the next one. Fedora 9 oopses occur with Linux 2.6.25 through 2.6.27, at which point they are replaced by oopses from Fedora 10. 2.6.27 is indeed the default kernel of Fedora 10. Thus, the initial appearance of oopses from one specific version of Fedora distribution is always associated with the default Linux kernel version that is shipped with the Fedora distribution. Furthermore, the association of Fedora distribution with the Linux kernel version evolves over time, as also indicated in Figure. 3.11. For instance, Fedora

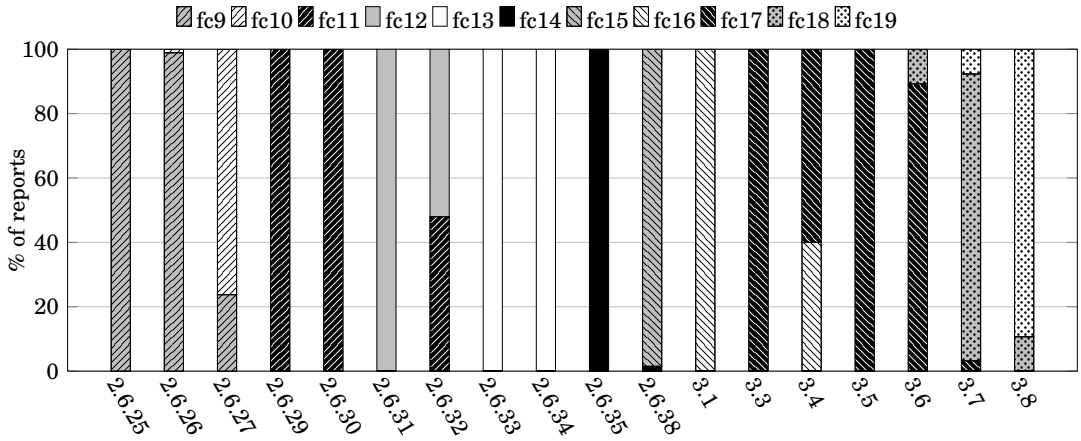

Figure 3.11: Prevalence of reports from different Fedora distributions for different versions (all versions for which there is at least one Fedora report are included)

11 initially was shipped with the Linux kernel version 2.6.29, its kernel then was upgraded to the version 2.6.30. Later it skipped the version 2.6.31 and was upgraded with the version 2.6.32. One can observe the similar trend with the Fedora 16, which started with the Linux kernel version 3.1 and stopped at the version 3.4.

Hardware architecture. Figure. 3.12 shows the number of bug and warning reports found in the repository for the 32-bit and 64-bit x86 architectures. We determine the architecture by the size of instruction addresses in the call trace and the values in the registers, whichever is available. We furthermore only consider reports for which a version and error type are available, representing 91% of the total number of reports. There are few reports for versions 2.6.18 and 3.9, and thus the information for these versions may not be representative.

We expect that the number of users of 32-bit architectures is declining, and thus there should be fewer reports from such architectures for more recent kernels. Indeed, starting with version 2.6.36, the rate of reports from 32-bit machines is almost always much lower than the rate of reports from 64-bit machines from versions prior to 2.6.36, particularly in terms of the rate of reports that represent bugs. Most Linux x86 code is shared between both the 32 and 64-bit architectures, and so this drop off is not likely to be due to an improvement in the quality of the 32-bit specific code. Thus, we conclude that the recent releases of Linux have fewer 32-bit reports simply because the 32-bit architecture has fewer users.

3.4 Features Related to Kernel Reliability

We now consider how to correctly interpret various features of an oops that may be useful in assessing kernel reliability.

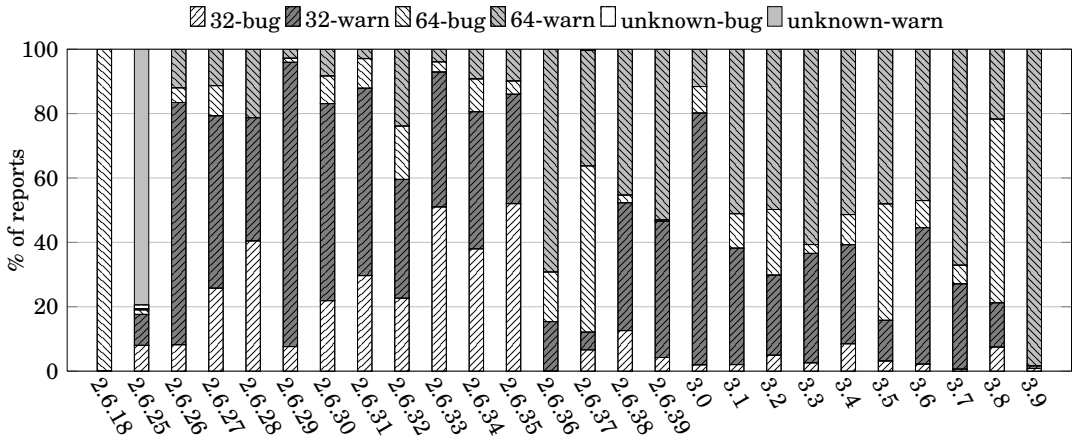

Figure 3.12: Prevalence of reports from 32 and 64 bit machines

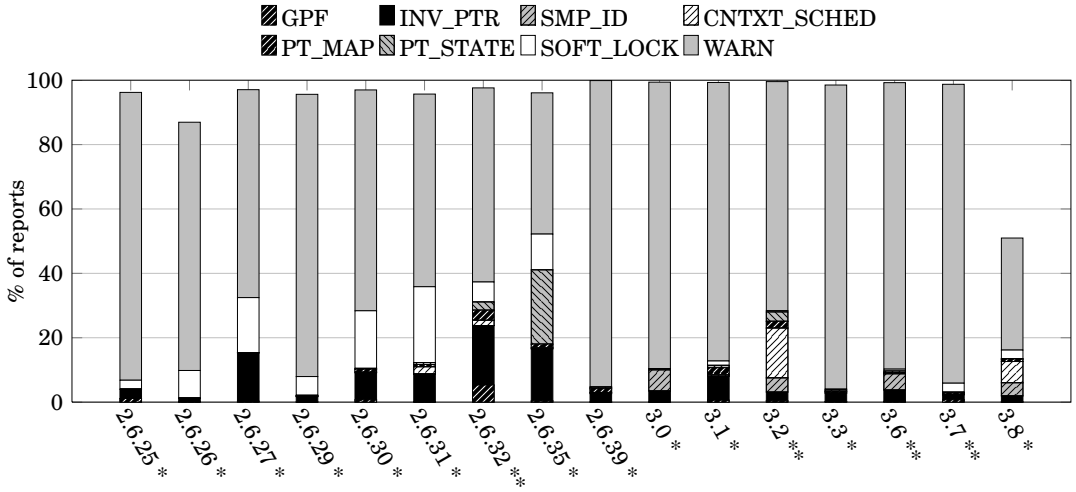

Figure 3.13: Prevalence of the 8 most common events (bugs or warnings)

Error types. Figure. 3.13 shows the frequency of various bugs and warnings, across the various versions, for the oopses specifying a version. In every case, warnings (`WARN`) make up the largest share of the oopses. Many of these warnings come from the same functions. Figure. 3.14 and 3.15 show that in some versions, a single function may be the source of over 90% of the warnings, for either the 32-bit or 64-bit architecture. We furthermore observe that it is useful to distinguish between these architectures, because some kinds of warnings that are prominent on one architecture are less common or absent on the other. For example, the warning in `default_send_IPI_mask_logical` is common on the 32-bit architecture, but the function is not used on the 64-bit architecture.

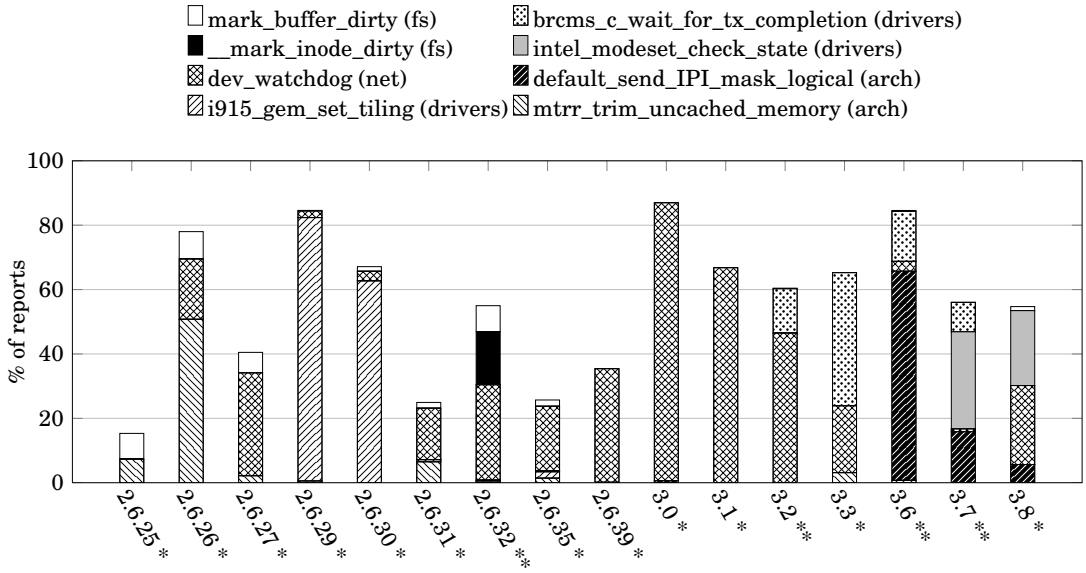

Figure 3.14: Top 8 warning-generating functions, 32-bit architecture

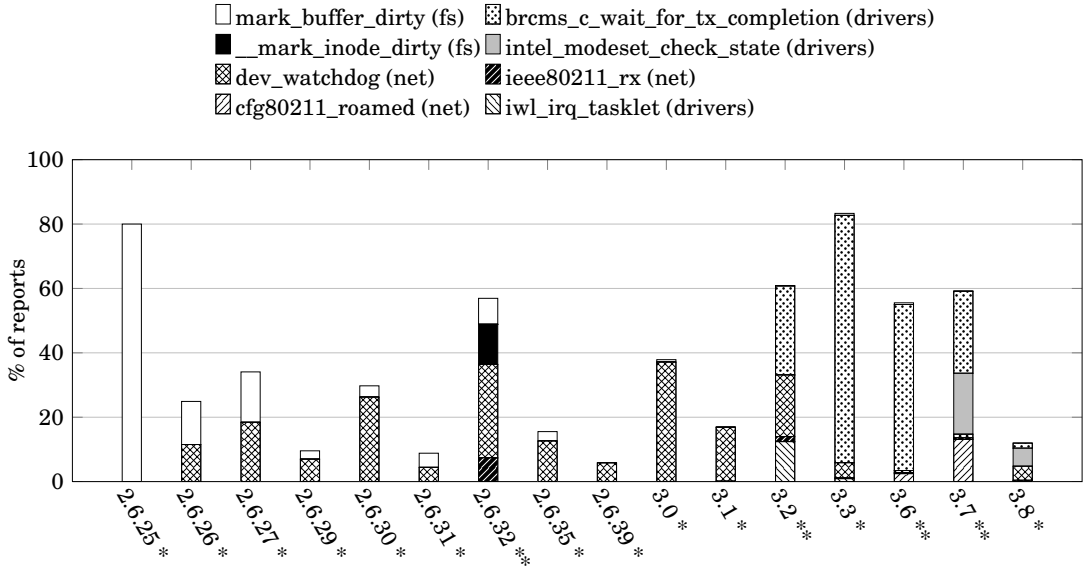

Figure 3.15: Top 8 warning-generating functions, 64-bit architecture

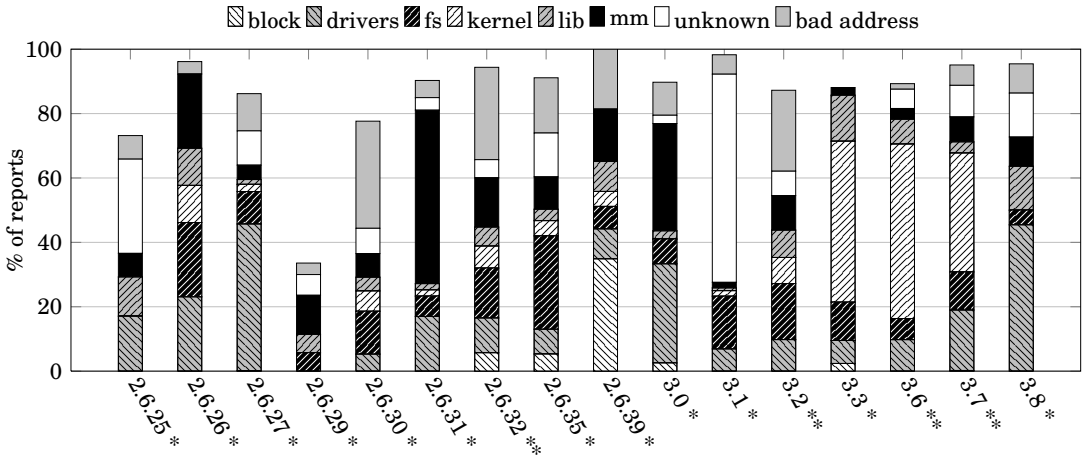

Figure 3.16: Top 6 services containing invalid pointer references

Call trace top origin. Several works on Linux code have identified drivers as a main source of faults, *e.g.*, based on the results of static analysis [32, 103]. A kernel oops repository provides an opportunity to determine whether drivers are the main source of errors encountered in practice. We consider invalid pointer references (`INV_PTR`), a common kind of serious error (Figure. 3.13), and study the service associated with each oops. We approximate the service as the top-level subdirectory containing the definition of the function in which the invalid reference occurs.

Figure. 3.16 shows the top 6 services in which invalid pointer references occur. Additionally, there are a number of cases where the service is unknown because the invalid reference occurs in an external function, that is not part of the kernel source tree (`unknown`), and where the service is unknown because the call to the function itself represents the invalid reference (`bad address`).

While driver functions do make up a large percentage of the functions in performing an invalid pointer reference, such errors occur in other services as well. Notably, in Linux 3.3, 3.6, and 3.7 there are many errors in `kernel` code. To understand why, we have also studied the names of the functions in which invalid pointer references most frequently occur. One such function is the primitive locking function `_raw_spin_lock`, which is defined in the `kernel` subdirectory. Because locking is such a basic functionality of the Linux kernel, it is not likely that the fault is in the definition of `_raw_spin_lock` itself, but rather in its caller, which may have attempted to lock an invalid lock. To identify the service associated with the caller, we search for the name of the calling function at the top of the call trace. We find that 74% of the failing calls to `_raw_spin_lock` come from file system functions.

Spikes. As shown in Figure. 3.9, there are two significant spikes in the oopses per day for Linux 3.6, on November 9, 2012 and late January, 2013. Such spikes can potentially dominate other data and distort assessments of reliability. As shown in Figure. 3.13, most of the reports for

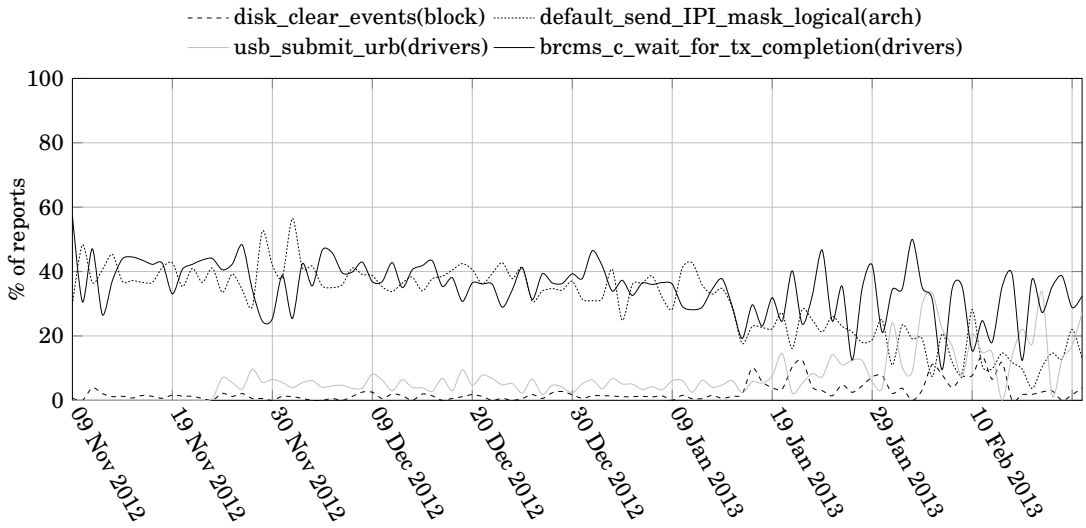

Figure 3.17: Common warnings by day for Linux 3.6

these versions are warnings. Thus, we study the frequency of warnings by day for Linux 3.6 to see if the number of oopses per day has an impact on the frequencies of these warnings.

Figure. 3.17 shows the rate of warnings from the most common warning-generating functions during the period in which there are the most oopses for Linux 3.6. While the rate of warnings from the most common warning-generating functions changes over time, there is no significant difference between November 9 or January 16 and the surrounding days. Thus, the spikes on these days affect the number of reports present, but not their relative number, and the information for these days can be safely combined with the rest.

Trigger action. We next consider the action that caused the kernel to be entered, which provides another perspective on which kernel services are error prone. Actions include those that are initiated by the kernel, such as booting and creating a kernel thread, those that are initiated by applications, such as system calls, and those that are initiated by devices, such as interrupts. To identify the trigger action, we analyze the bottom of the call trace. In doing so, we must take into account stale pointers, as was described above. But we also must take into account the particular structure of Linux call traces.

For both 32-bit and 64-bit x86 architectures, Linux manages a linked list of stacks, comprising a process stack, an interrupt stack, and, on 64-bit x86 architectures, a set of exception stacks.[3] For the 32-bit x86 architecture, there is an explicit annotation of an interrupt stack only if both an interrupt stack and a process stack are present. Thus, there is an ambiguity; if an interrupt occurs when no process is running in the kernel, then there is only one stack and no delimiter.

[3]On the 64-bit x86 architecture, a serious interrupt is referred to as an exception.

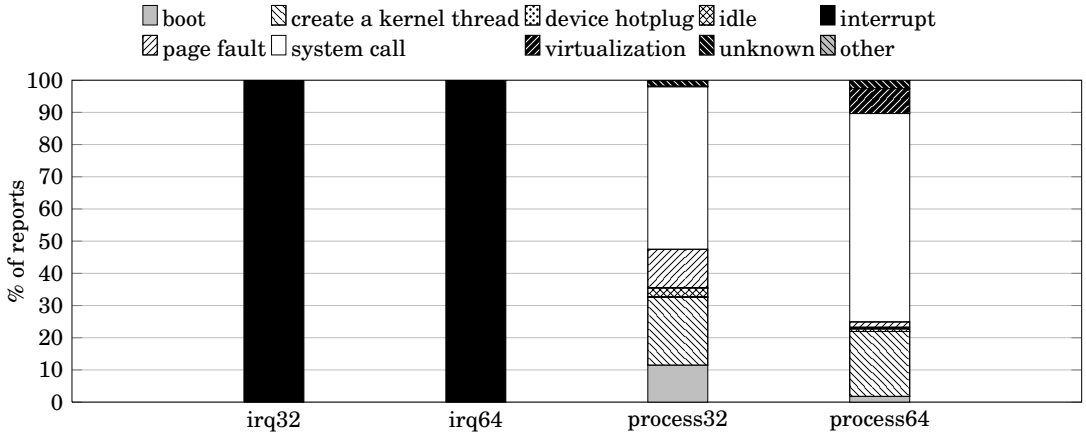

Figure 3.18: Stack bottoms of interrupt and process stacks

We thus classify a 32-bit call trace containing only a single stack as representing an interrupt stack if the bottom function of the call trace is an interrupt function, such as `common_interrupt` or `apic_timer_interrupt`. For the 64-bit x86 architecture, interrupt stacks are delimited by `IRQ` and `EOI` (End Of Interrupt) and exception stacks are delimited by the name of the exception and `EOE` (End Of Exception). There is no ambiguity in this case.

To identify the action that triggered an error, we consider the bottom-most non-stale function in the current stack, if any, or if there is no non-stale pointer, then the bottom-most stale function in the current stack. This strategy covers 89% of the reports, excluding those for which there is no call trace or for which the architecture cannot be determined. We have manually classified these functions according to what kind of action they represent: system boot, creation of a kernel thread, the idle process, an interrupt, a page fault, a system call, or support for virtualization. Figure. 3.18 shows the results for interrupt and process stacks for the 32-bit and 64-bit x86 architectures, considering only those functions that occur at least 50 times. We have only 420 occurrences of an exception stack, and none of the stack bottoms satisfied this threshold.

Most of the results are as could be expected: the trigger action for an interrupt stack is always an interrupt, and the most common trigger action for a process stack is a system call. We do have a small anomaly, that is not visible in the graph, for 64-bit process stacks. Although the interrupt stack is in principle unambiguous for the 64-bit x86 architecture, we have over 350 cases where an interrupt function appears at the bottom of what seems to be a 64-bit x86 process stack. All of these instances come from Suse. Indeed, only 5 Suse reports out of the 1139 having a call trace have an interrupt stack delimiter, and thus we conjecture that the Suse oops reporting tool may eliminate them.

Table 3.2: Taint types

Loaded code:	**Specific errors:**
P: Proprietary module loaded.	M: Machine check exception.
G: No proprietary module loaded.	B: System has hit bad_page.
F: Module forcibly loaded.	U: Userspace-defined naughtiness.
R: Module forcibly unloaded.	A: ACPI table overridden.
C: drivers/staging modules loaded.	I: Severe firmware bug.
O: Out-of-tree modules loaded.	S: SMP with non-SMP CPU.
Generic errors:	
D: Previous die.	W: Previous warning.

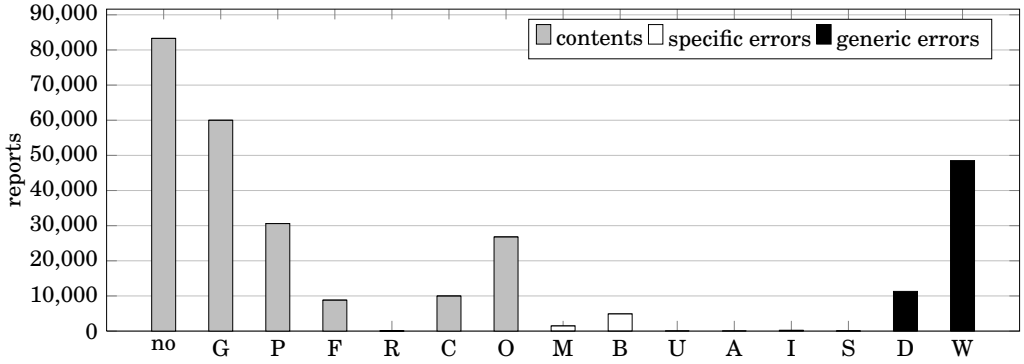

Figure 3.19: Number of occurrences of individual taint bits

Impact of taint. Because the Linux kernel is intended to be a long-running system and because it must meet stringent performance requirements, it cannot keep available in memory an unbounded history of its actions. Thus, an oops primarily provides an instantaneous picture of the state of the kernel. Nevertheless, when certain kinds of events have occurred in the kernel execution, the kernel may have been left in an unreliable state. Thus, a small amount of historical information is recorded, known as *taint*, which amounts to a one-word bit map recording the previous occurrence of various events. The events considered involve the inclusion of suspect code in the kernel, and the previous occurrence of specific and generic errors. We list all types of taint bit in Table.3.2. The presence of taint can reduce the credibility of an oops as a record of an error in the kernel.

Figure. 3.19 shows the number of occurrences of the individual taint bits across the reports. Only reports that have taint information are included, amounting to 93% of the reports. Almost half of these reports have no taint (`no`). For these oopses, there is the greatest likelihood that the source of the problem is captured in the oops, rather than depending on some previous action. Of the remainder, two thirds (`G`) do not involve proprietary modules, meaning that maintainers can reconstruct the execution environment using code that is freely available. Of those that do not involve proprietary modules, 45% do involve external modules (`O`), implying some extra work for the maintainer to find the relevant code. Finally, 6% of the reports containing taint information derive from kernels that include so-called “staging” drivers (`C`). These immature drivers are integrated into the kernel so that they can keep up with kernel changes as they mature. The

kernels involved may be development kernels. Few of the reports come from kernels that have previously experienced the specific errors represented by the taint field (`M` through `S`); only the numbers of machine check exceptions (`M`) and bad pages (`B`) are non-negligible. Finally, 6% of the reports containing taint information come from kernels that have previously experienced a serious error (`D`), and 28% come from kernels that have previously experienced a warning (`W`).

3.5 Threats to Validity

The overall goal of our study is to identify the potential threats to validity in using kernel oopses as an indicator of Linux kernel reliability. Our study itself, however, may suffer from threats to validity, related to our interpretation of the studied data. To ensure the representativeness of the data, the author studied a collection of data over a period of 8 months. To ensure the validity of the analysis, the author also studied the source code of the user-level oops-reporting tools (*i.e.* *ABRT* from Fedora and *kerneloops* from Debian), and duplicated the process of generation and submission of kernel oopses to the repository.

3.6 Conclusion

The Linux kernel, due to its wide adoption, has been studied extensively in previous works [98, 102, 117, 132]. In this chapter, rather than studying the source code of the Linux kernel, the author studied a specific error report (*i.e.* kernel oops) generated from the Linux kernel. The author has provided a comprehensive analysis of the Linux kernel oopses available in the recently revived oops repository maintained by Red Hat. The analysis has revealed the information available in the repository, but also the challenges involved in interpreting it. As studies of repositories such as those of Windows and Mozilla have shown, an oops repository has great potential to shed light on the reliability of the Linux kernel, as well as to help study development practices. The results can serve as a guideline for the correct interpretation of the results of future research studies.

CHAPTER 4

KERNEL OOPS DEBUGGING

Debugging a kernel oops remains a daunting task for the Linux kernel developers. Indeed, Linus Torvalds once teased, "*How to track down an Oops...The main trick is having 5 years of experience with those pesky oops messages ;-)*" [123]. By *debugging*, here we restrict ourself to a smaller and yet critical goal — locating the *offending line*, instead of finding the root cause of a bug. The reason is twofold: I). a kernel oops does not necessarily correspond to a bug which is a more abstract concept of software defect. It is simply an instance of the manifestation of a software error. Multiple bugs might trigger the same kernel oops. II). the offending line is well-defined and useful information for a Linux kernel developer to initiate the process of finding the root cause of a bug, while the root causes of bugs vary from case to case and there is no uniform representation of root cause.

Yet, even this goal is not easy to achieve. Since a kernel oops is intended to provide only a compact status information for debugging, lots of information is missing from there. Given the limited information provided by a kernel oops, a Linux developer may have difficulty to reconstruct the crashing image of the Linux kernel, not to mention reproducing the crashing scenario. Therefore, Linux developers tend to rely on the cooperation of users when debugging a reported kernel oops. As illustrated in Chapter 1, for a kernel oops report [28] from the Linux kernel bugzilla, it took 16 rounds of comment changes between a developer and several users to finally figure out the offending line of the kernel oops.

In contrast to the bugzilla case, in a more common scenario, when a developer finds an oops in the kernel oops repository that may be relevant to his code, there is no associated user that the developer can contact. This compounds the difficulty of locating the offending line. Several options remain. If the developer can deduce the exact version and distribution from the oops, then he may be able to retrieve a precompiled version of that kernel with debugging information

from that distribution, if such a kernel is available. Alternatively, the developer can compile the kernel file containing the crashing function from the same Linux version, to try to recreate the victim kernel's binary code. If either of these is successful, the developer can load the kernel code into the debugger gdb and request the source code line corresponding to the instruction pointer mentioned in the oops. A second option is to disassemble the small snippet of binary code surrounding the crash site, that is found in oopses corresponding to the more severe types of bugs, and then match the result against the assembly code of the crashing function. A third option is to analyze the semantics of the instructions found in the disassembled code snippet, to relate this semantics to the source code.

In this chapter, we argue that the second approach, of matching the disassembled code snippet to the assembly code of a locally compiled kernel, is the most promising. The first approach, using the instruction pointer, is lightweight but brittle, especially in the common case where it is not possible to retrieve the distribution's original debugging kernel. In contrast, the code snippet used in the second approach provides more context information, and is thus more tolerant to variations in locally generated code. The third approach, relating the semantics of the disassembled code snippet directly to the source code, is very difficult, due to the different levels of abstraction involved. The second approach nevertheless has the disadvantage that the matching process is tedious and error prone, especially for the many large kernel functions. To make the second approach practical, we propose to automate the matching process, based on a variant of *approximate string matching*, as used in bioinformatics [121].

The rest of the chapter is organized as follows. Section 4.1 gives an illustration of the state-of-the-art approaches to obtaining the offending line. Section 4.2 presents the outline of our matching algorithm. Section 4.3 instantiates this algorithm with some domain-specific design choices. We then evaluate our solution in Section 4.4. Finally, we conclude this chapter in Section 4.6.

4.1 State-of-the-Art

In this section, we revisit the content of a kernel oops presented in Chapter 3 and explain the existing approaches to finding the offending line. We then present some statistics that illustrate the difficulty of the latter process.

4.1.1 Revisit Kernel Oops

A kernel oops documents the internal state of the Linux kernel at the time of a failure. It is represented in plain text, and comprises a number of fields. Each field represents one aspect of the system state in the form of a key-value pair.

We have already given a detailed description of the contents of a kernel oops in Section 3.1.1. Here we revisit some of the key fields (marked in box) of a kernel oops shown in Figure 4.1,

```
1  BUG: unable to handle kernel NULL pointer dereference at 00000008
2  IP: [<c10fb360>] xfs_da_do_buf+0x4da/0x5e1
3  Oops: 0000 [#1] PREEMPT SMP
4  Modules linked in: nfs lockd nfs_acl sunrpc w83627hf ...
5  Pid: 2725, comm: rm Tainted: P 2.6.36-gentoo-r5 #10
6  EIP: 0060:[<c10fb360>] EFLAGS: 00010246 CPU: 0
7  EAX: 00000001 EBX: f60da400 ECX: fbd5a730 EDX: 00000000 ...
8  Stack: 0012c096 00000000 f64cabc0 f64ca5dc d853fb40 ...
9  Call Trace:
10  [<c1020717>] ? get_parent_ip+0xb/0x31
11  [<c102086e>] ? sub_preempt_count+0x7c/0x89
12  [<c111f85e>] ? xfs_remove+0x1b3/0x2e0 ...
13 Code:  f0 00 c7 45 b8 00 00 00 00 74 13 8b 4d 18 8d 55 f0 b8 01 00
    00 00 e8 06 fa ff ff 89 45 b8 83 7d 14 01 0f 85 82 00 00 00 8b 55
    <8b>  4a 08 8b 51 08 8b 01 0f c8 86 f2 0f b7 d2 81 fa ee fb 00 00
14 ---[ end trace 118398ff1b25f91d ]---
```

Figure 4.1: XFS kernel oops

which are used by the standard approaches to find the offending line. Particularly, we highlight a previously neglected field called ***code snippet*** that is essential to our approach.

Oops description In our example, the oops was triggered by NULL-pointer dereference, as indicated by the brief description of the cause of the oops at Line 1.

Error site The *IP* (Instruction Pointer, line 2) field indicates the name (`xfs_da_do_buf`) of the function being executed at the time of the oops, *i.e.*, the crashing function, the binary code offset (`0x4da`) at which the oops occurred, and the binary code size (`0x5e1`) of the crashing function.

Kernel version A string (`2.6.36-gentoo-r5`) following the value of the attribute *Tainted* (line 5) indicates the Linux kernel version (`2.6.36`, maintained between October 2010 and February 2011) from which the kernel oops was generated. The string may also indicate the distribution of the Linux kernel, here Gentoo.

Code snippet Some kernel oopses, for the more severe faults, contain a binary code snippet (line 13) indicating the 64 bytes of binary code surrounding the error site (43 bytes before the trapping instruction and 20 after), regardless of instruction boundaries. Within this snippet, the first byte of the instruction that causes the crash is marked by <>. We refer to this instruction as the *trapping instruction*.

4.1.2 Pinpointing the Offending Line

We now consider in more detail the three options for pinpointing the offending line that were proposed in the introduction, in terms of the example in Figure 4.1. We place ourselves in the situation where a kernel with debugging information corresponding to the offending instruction is not available, as finding such debugging kernels is difficult for versions that do not correspond to a current release of the given distribution, and even if possible requires somewhat obscure distribution-specific knowledge.

Gdb To use gdb on the instruction pointer found in the kernel oops, we first need to obtain binary code with debugging information for the file containing the crashing function. For this, we download the kernel version corresponding to the version (2.6.36) listed in the kernel oops from the mainline kernel repository `kernel.org`. We then create a default configuration file, using the command `make ARCH=i386 defconfig` because the oops contains 32 bit addresses. In the configuration file, to obtain debugging information, we then manually select the option `CONFIG_DEBUG_INFO` and unselect the option `DEBUG_INFO_REDUCED`. We then use the local version of gcc (in our case, gcc 4.7.2 (Debian 4.7.2-5)) to compile the file containing the crashing function, using the command `make ARCH=i386 fs/xfs/xfs_da_btree.o`. Finally, we load the resulting object file into gdb and obtain the offending line with the command `list *(xfs_da_do_buf+0x4da)`.

Having followed this procedure, we obtain a line that is five lines away from the one indicated in the bugzilla discussion. There are furthermore several lines at which a NULL pointer dereference could occur between the line identified by gdb and the actual offending line. This approach is clearly too brittle for practical use.

Assembly code matching For this approach, we first obtain a locally compiled debugging kernel, as above. We then use the Linux kernel script `decodecode`, with the environment parameter `AFLAGS=--32|--64` indicating the architecture, to disassemble the code snippet. Next, we use gdb with the command `disassemble xfs_da_do_buf` to disassemble the crashing function. We must then manually match the two sequences of assembly code to find the offset of the instruction in the locally compiled crashing function that most probably matches the trapping instruction in the code snippet. This offset can then be used with gdb as in the previous case, to get the offending line.

In the case of the oops shown in Figure 4.1 the locally compiled crashing function consists of 459 instructions, as compared to the 19 instructions in the code snippet. The large size of the crashing function makes the matching process tedious and error prone. Indeed, an exact match is not likely, due to the use of different compilers or compiling options, which causes differences in both the choice of instructions and the order in which they appear. These problems can be

```
1 int elv_may_queue(struct request_queue *q, int rw)
2 {
3       struct elevator_queue *e = q->elevator;
4       if (e->ops->elevator_may_queue_fn)
5             return e->ops->elevator_may_queue_fn(q, rw);
6       return ELV_MQUEUE_MAY;
7 }
```

Figure 4.2: The source code of the crashing function `elv_may_queue`

```
code snippet                  crashing function
<+0>  push %ebx               <+0>  push %ebp
<+1>  mov %eax,%ebx           <+1>  mov %esp,%ebp
---                           <+3>  call 0x1414 (<+4>)
<+3>  mov 0xc(%eax),%eax      <+8>  mov 0xc(%eax),%ecx
<+6>  mov (%eax),%eax         <+11> mov (%ecx),%ecx
<+8>  mov 0x38(%eax),%ecx     <+13> mov 0x38(%ecx),%ecx
<+11> xor %eax,%eax           ---
<+13> test %ecx,%ecx          <+16> test %ecx,%ecx
<+15> je 0x38 <+21>           <+18> je   0x1428 <+24>
<+17> mov %ebx,%eax           ---
<+19> call *%ecx              <+20> call  *%ecx
<+21> pop %ebx                <+22> pop   %ebp
<+22> ret                     <+23> ret
---                           <+24> xor   %eax,%eax
---                           <+26> pop   %ebp
---                           <+27> nop
<+23> lea 0x4c(%edx),%eax     <+28> lea 0x0(%esi,%eiz,1),%esi
---                           <+32> ret
```

Figure 4.3: An example of assembly code matching

somewhat alleviated if the instruction pointer indicates that the trapping instruction is very near the beginning or end of the function. Our example, however, does not have this property.

At the other end of the spectrum, we consider an oops derived from the much smaller function shown in Figure 4.2. Figure 4.3 shows the correspondence between the code snippet in the oops and the assembly code obtained from locally compiling the crashing function. By comparing the sequence of opcodes, we can see that the trapping instruction at offset <+8> in the code snippet corresponds to the instruction at offset <+13> in the locally compiled crashing function. The debugging information then indicates that the instruction at offset <+13> of the crashing function originates from the statement at line 4 in Figure 4.2, which is the offending line. During the matching process, some gaps, marked by ---, are required, and we can see that the compilers

have taken different decisions in compiling the conditional test, marked with arrows at offsets <+15> and <+18>, respectively. These issues complicate the matching even for a small example.

Semantics matching For this approach, we use decodecode as above to obtain the assembly code of the code snippet and download the kernel version mentioned in the oops, but do not compile it. For simplicity, we illustrate this approach using only our second example (Figures 4.2 and 4.3). We reason as follows: I) The trapping instruction at the offset <+8> is a memory access operation, due to the operand 0x38(%eax), which implies that the offending line involves a pointer dereference. II) Right above the trapping instruction, there is a similar instruction at the offset <+6> that operates on the same register as the trapping instruction. Thus, the offending line likely contains two successive pointer-dereferences, *i.e.*, it has the form $a \rightarrow b \rightarrow c$, for some expression a, and some fields b and c. III) Following the trapping instruction, there is a branching test at the offset <+15> which should correspond to a conditional test in the source code. IV) Furthermore, the result of the trapping instruction is stored in the register %ecx, which is used later at offset <+19> as a function pointer (call *%ecx). Thus, the offending line should evaluate to a function pointer for a following invocation. Line 4 in the crashing function is the only one that satisfies all of the above properties, and is indeed the offending line.

Although successful in this case, this approach is clearly not systematic enough for frequent use, particularly for large or complex crashing functions, such as the one associated with Figure 4.1. Nevertheless, we use this approach, combined with some manual matching of the assembly code to validate our results obtained in Section 4.4.

4.1.3 Properties of Oopsing Functions

To better understand the problems confronting a kernel developer, we have studied various properties of the oopses collected during the first 8 months of the existence of the kernel oops repository.

Data set Between September 1, 2012 and May 1, 2013, the kernel oops repository received 187,342 kernel oopses. Many are warnings, and thus only 22,316 contain a code snippet. Our approach requires the presence of both a code snippet and a function name. 3263 of the code snippets contain only “Bad EIP value” or “Bad RIP value” indicating an invalid instruction pointer. Of the remaining 19,053 oopses, 897 do not give a name for the crashing function, only an address, again indicating an invalid instruction pointer. Finally, the kernel oops repository contains many duplicate reports [48]. Of the oopses containing both a function name and a code snippet, 5192 have a unique combination of function name, crashing instruction offset, function size, and code snippet. We consider only these oopses in the rest of this section. We summarise the above classification about the data set in Table. 4.1.

Table 4.1: Statistics about the data set

category	**count**	**reduction**
all oopses	187,342	
		165,026
with code snippet	22,316	
		3,263
with valid instruction pointer	19,053	
		897
with crashing function name	18,156	
		12,864
not duplicated	5192	

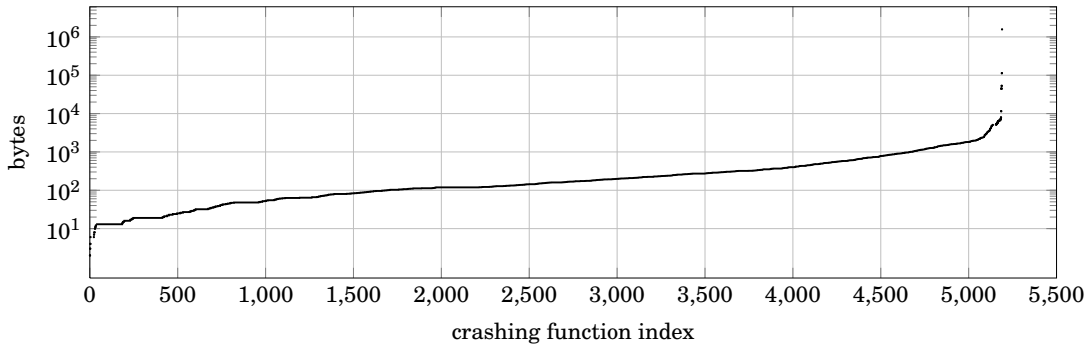

Figure 4.4: Function size for unique oopses containing both a code snippet and function name

Size of the crashing function As the size of the crashing function increases, it becomes more difficult to scan through its assembly code to find instructions that look similar to the disassembled code snippet. Figure 4.4 shows the sizes in bytes of the crashing functions indicated in our oopses, ordered from smallest to largest.[1] This figure shows one point per oops, even though multiple oopses may refer to the same function. 20% of the unique oopses involve functions containing more than 500 bytes, making manual scanning for a close match to a 64-byte code snippet impractical. Three oopses refer to a function that contains over a million bytes. This is a large memory region used by a kernel module added by VirtualBox, and is not part of the mainline kernel.[2]

Compiler effects A major source of variation in the sequence of instructions generated by compiling a kernel is the version of the compiler used. To measure this effect, we consider three compilers: gcc 4.6.1 (Ubuntu/Linaro 4.6.1-9ubuntu3), gcc 4.7.2 (Debian 4.7.2-5), and gcc

[1] We use the number of bytes indicated in the oops; the number of instructions is not available.

[2] https://forums.virtualbox.org/viewtopic.php?f=7&t=30037

Table 4.2: Distribution of sampled crashing functions

category	not universal	not included in kernel conf	not generate obj files	the rest
percentage	25%	18%	3%	54%

4.8.1 (Ubuntu/Linaro 4.8.1-10ubuntu9). For simplicity, we test these compilers on a single version, Linux 3.7, released in 2012. For this kernel, we generate a configuration file using `make ARCH=x86_64 defconfig`. Linux 3.7 represents 12% of our considered oopses, and is the second most common version mentioned. The most common version, representing 57% of the oopses, is Linux 2.6.32, first released in 2009. Our compilers, however, generate many warnings on the Linux 2.6.32 code, and we want to avoid comparing invalid code.

Our single tested kernel version and configuration may not cover all the crashing functions of our 5192 oopses. There are 1557 such functions, of which 25% are either not defined in Linux 3.7, are only defined in architecture-specific code (`arch` directory) for an architecture other than x86, or are only defined in header files, which do not generate an independent object file in which we can find the binary code definition. Finally, for 18% of the 1557 functions, the defining file is not included in our kernel configuration, and for 3% of the 1557 functions, at least one compiler does not generate code, perhaps due to inlining. This leaves just over 850 distinct crashing functions whose assembly code we can compare. We summarise the above statistics in Table. 4.2.

For each compilation result, we first obtain the assembly code, using the command `objdump -disassemble`, and then rewrite the output to abstract away all information except the opcode, any constants, and any constant offsets used in indirect references. This eliminates the effect of compiler choices such as register names and branch offsets, and focuses on information that is likely to stand out as the developer scans the code. To estimate the difference between two compilation results we apply the unix tool `diff` to these abstracted versions, and then divide the sum of the number of differences by twice the number of instructions found in the smaller compiled function, to normalize the results. As a baseline, we also compute the difference in size between the smaller and larger compiled functions as a percentage of the size of the smaller compiled function. Finally, we distinguish between small functions, for which the smaller compiled function contains 50 instructions or less, medium functions, for which the smaller compiled function contains 250 instructions or less, and large functions, for which the smaller compiled function contains more than 250 instructions. The number of functions in each category varies slightly depending on which compilers are being compared.

The results are shown in Table 4.3. In each case, the comparison on instruction opcodes and operands shows a much greater rate of change than the comparison on code size. Thus, many changes must be within the shared part of the code; there is not, for example, simply a new sequence of code added at the end of the function. Furthermore, we see that the rate of changes increases as the function size increases and as the distance between the compiler versions increases. All these variations make manual matching complex.

Table 4.3: Compiler comparison.

	1 gcc version						2 gcc versions		
	4.6.1 → 4.7.2			4.7.2 → 4.8.1			4.6.1 → 4.8.1		
function size	instr diff	sz diff	# fns	instr diff	sz diff	# fns	instr diff	sz diff	# fns
small	14%	8%	353	14%	4%	350	23%	11%	354
medium	17%	4%	404	21%	4%	402	28%	6%	397
large	32%	3%	102	32%	4%	102	40%	5%	103

Instr diff is the rate of difference in the instructions. Sz diff is the rate of difference in the function sizes. #fns is the number of functions that are small, medium, or large, respectively.

4.2 Automating Assembly Code Matching

Our goal is to automate the matching of the disassembled code snippet against the disassembled locally compiled crashing function, as previously illustrated in Figure 4.3. We take inspiration from *approximate sequence matching*, used in bioinformatics [115, 121]. We begin with some definitions, and then propose a matching algorithm that focuses on each possible counterpart to the trapping instruction and its surrounding context. This matching algorithm uses an algorithm for matching a code sequence against a prefix of another code sequence, which we present next. Finally, we consider the complexity of the complete approach. Some design decisions related to the particular properties of assembly language instructions are deferred to Section 4.3.

4.2.1 Definitions

We refer to disassembled code snippet as the *code snippet*, C, and the disassembled locally compiled crashing function as the *local function*, L. We first define some properties of sequences, and then describe sequence matching:

Definition 4.1. *For a sequence S, let $S[i]$ denote the element at the position i, and $|S|$ denote the length of S. We also define a special element called a* gap*, denoted $\perp$.*

Definition 4.2. *For a sequence S of length n and any $1 \le i \le j \le n$, $\widehat{S} = S[i...j]$ is a **substring** of S and $\widehat{S} = S[1...j]$ is a **prefix** of S.*

Definition 4.3. *For sequences S_1 and S_2, we define an **anchored alignment** of S_1 with S_2, written* Align(S_1, S_2)*, as any (S'_1, S'_2) where:* I) *S'_1 and S'_2 are sequences that may contain gaps.* II) *$|S'_1| = |S'_2|$.* III) *Removing gaps from S'_1 leaves S_1.* IV) *Removing gaps from S'_2 leaves a* prefix *of S_2.*

Anchored alignment is a variant of the ***global alignment*** and ***local alignment*** used in bioinformatics [115, 121]. Global alignment matches two complete sequences, while local alignment matches subsequences of those sequences. Anchored alignment matches a complete sequence against a prefix of another sequence, with gaps allowed. Figure 4.5 shows an example.

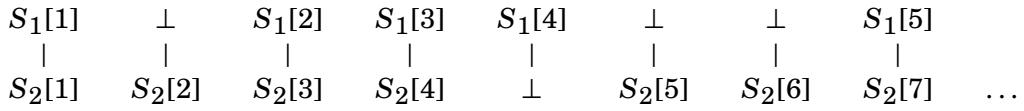

Figure 4.5: *Align*(S_1, S_2), where $|S_1| = 5$ and $|S_2| \geq 7$

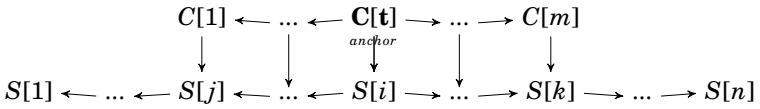

Figure 4.6: Anchored sequence matching schema

In general, to find the offending line, it is not sufficient to use any match; instead, we want one that is considered to be best according to some scoring function. Indeed, the challenge in performing alignment is to determine where to place gaps to obtain the highest possible score.

Definition 4.4. *For elements s_1 and s_2, let the function* **score**(s_1, s_2) *be a measure of the similarity between s_1 and s_2.*

The precise definition of the *score* function is orthogonal to the problem of matching, and is deferred to the next section. Nevertheless, we require that a match with a gap have a negative score, to make it unfavorable to match a gap with itself.

Definition 4.5. *For sequences S'_1 and S'_2 of length n, which may contain gaps, we define the* ***similarity*** *between S'_1 and S'_2, as* $\mathrm{Sim}(S'_1, S'_2) = \sum_{i}^{n} \mathrm{score}(S'_1[i], S'_2[i])$.

Definition 4.6. *For sequences S_1 and S_2, let Θ be the set of all possible anchored alignments between S_1 and S_2, i.e.,* $\mathrm{Align}(S_1, S_2)$*. Then, the* ***optimal anchored alignment*** *of S_1 with S_2, written* $\mathrm{Align}_{opt}(S_1, S_2)$*, is the one of* $\{\theta \in \Theta \mid \forall \theta_1 \in \Theta, \mathrm{Sim}(\theta) \geq \mathrm{Sim}(\theta_1)\}$.

4.2.2 Anchored Sequence Matching

We define anchored sequence matching to select a *counterpart instruction* from the local function, that matches well against the trapping instruction, taking the contexts of these instructions into account. First, we select a set of *anchor points*, likely candidates for the counterpart instruction from the local function. Next, starting from each *anchor point*, we compute the optimal match between the preceding and following fragments of the code snippet and of the local function, respectively. Finally, we select the anchor point with the maximal overall similarity value as the counterpart instruction.

Algorithm 3 presents the anchored sequence matching algorithm. Lines 2-6 perform some initializations, including splitting the code snippet into three parts: the part before the trapping instruction, *C_left*, the trapping instruction, *trap*, and the part following the trapping instruction,

C_right. *C_left* is reversed, because we want to match the sequence to the left of the trapping instruction backwards from the trapping instruction (*cf.* Figure 4.6). Lines 8-22 then iterate over the possible anchor points. For each one, the local function is split into *L_left*, *anchor*, and *L_right*, with *L_left* reversed (lines 9-12), analogous to *C_left*. Optimal anchored alignment is then used via the function **align** to match *C_left* against *L_left*, and *C_right* against *L_right*, and to obtain the resulting similarity score (lines 13-14). We also compute the similarity score of the trapping instruction and the chosen anchor point (line 15). If the sum of these scores is greater than the best score recorded so far, then the current anchor point is recorded as the best match (lines 16-21). The final result is then the best collected anchor point (line 23).

Algorithm 3 The anchored sequence matching algorithm

```
 1: function ASM(List C, List L, List anchors, Int t)
 2:     best_anchor ← −1
 3:     max_score ← min_value
 4:     C_left ← reverse(C[1..t − 1])
 5:     trap ← C[t]
 6:     C_right ← C[t + 1..C.length]
 7:
 8:     for i = 1 → anchors.length do
 9:         L_left ← reverse(L[1..t − 1])
10:         anchor ← L[i]
11:         L_right ← L[t + 1..L.length]
12:
13:         left_score ← align(C_left, L_left)
14:         right_score ← align(C_right, L_right)
15:         anchor_score ← score(trap, anchor)
16:         total_score ← left_score + right_score + anchor_score
17:
18:         if (total_score > max_score) then
19:             max_score ← total_score
20:             best_anchor ← i
21:         end if
22:     end for
23:     return best_anchor
24: end function
```

4.2.3 Anchored Alignment

Anchored sequence matching uses anchored alignment, $Align(\widehat{C},\widehat{L})$, to match the code snippet substrings $\widehat{C}$ before and after the trapping instruction against the context $\widehat{L}$ of each proposed anchor point. Algorithm 4 shows a straightforward recursive implementation. For each pair of nonempty sequences this algorithm takes the maximum value produced by three possibilities: matching the heads of each sequence with each other and the tails of each sequence recursively with each other, or matching the head of one sequence with a gap and the tail of that sequence recursively with the other sequence. The remaining lines address the empty sequence cases. If

the code snippet substring is empty (line 3), the result is 0, reflecting the fact that it need only be matched against a prefix of the local function substring. On the other hand, if the local function substring is empty (line 4), the result is the product of the length of the remaining code snippet substring and the gap score, effectively matching the remaining code snippet substring to gaps.

Algorithm 4 A recursive anchored alignment algorithm

```
1:  function align_rc(List Ĉ, List L̂)
2:      match (Ĉ, L̂) with
3:      | ([ ], _) → 0
4:      | (_, [ ]) → |Ĉ| × gap_score
5:      | (c::ct, l::lt) →
6:          let gap_score  = score(⊥, ⊥) in
7:          let head_score = score(c, l)  in
8:          let no_gap = align(ct, lt) + head_score in
9:          let cs_gap = align(ct, L̂) + gap_score in
10:         let lk_gap = align(Ĉ, lt) + gap_score in
11:             return max(no_gap, cs_gap, lk_gap)
12: end function
```

The recursive algorithm constructs a three-way tree of execution, in which each branch has height at most $|\widehat{C}| + |\widehat{L}|$. $|\widehat{C}|$ has fixed size, and except when the local function is very small, $|\widehat{C}| \leq |\widehat{L}|$. The complexity is thus $O(3^{|\widehat{L}|})$.

The recursive algorithm recomputes the scores of many possible alignments. To avoid this inefficiency, we use dynamic programming [25], as shown in Algorithm 5. This algorithm populates an array *dpm* of size $(|\widehat{C}|+1) \times (|\widehat{L}|+1)$. The value stored in $dpm[i][j]$ represents the similarity of the best alignment between $\widehat{C}[1..i]$ and $\widehat{L}[1..j]$. As anchored alignment requires matching the complete code snippet substring but only a prefix of the local function substring, we take the best value found anywhere in the row $dpm[|\widehat{C}|][*]$ as the result. This is in contrast with the dynamic programming implementation of global alignment [115, 121], in which the iterative computation is the same, but the result is $dpm[|\widehat{C}|][|\widehat{L}|]$.

The dynamic programming algorithm (shown in Algorithm 5) starts by initializing the zero column of every row with a multiple of the gap score (lines 6-8), reflecting the need to match the elements of the code snippet substring against gaps when the local function substring runs out. It then initializes the zero row of every column with a multiple of the gap score (line 10-12), reflecting the cost of pushing the initial match of the code snippet substring past a prefix of the local function substring, which requires matching the elements of that prefix to gaps. The remainder of the algorithm iterates over the elements of the array, starting from the elements with the smallest indices, representing the early parts of each sequence and moving towards the elements with the largest indices, representing the later parts of the sequences. For each pair of offsets, the algorithm considers the possibilities that the sequence elements represented by the current position match (line 18), or that one or the other is matched against a gap (lines 19-20). In each case, the matching score or the gap score is added to the previously computed value found

Algorithm 5 Anchored alignment by dynamic programming

```
1: function Align_dp(List Ĉ, List L̂)
2:     // init matrix with zero
3:     dpm ← make_matrix(0..|Ĉ|, 0..|L̂|, 0)
4:     gap_score ← score(⊥, ⊥)
5:
6:     for i = 1 → |Ĉ| do
7:         dpm[i][0] ← i * gap_score
8:     end for
9:
10:    for j = 1 → |L̂| do
11:        dpm[0][j] ← j * gap_score
12:    end for
13:
14:    for i = 1 → |Ĉ| do
15:        for j = 1 → |L̂| do
16:            head_score ← score(Ĉ[i], L̂[j])
17:            dpm[i][j] ← max (
18:                dpm[i − 1][j − 1] + head_score,
19:                dpm[i − 1][j] + gap_score,
20:                dpm[i][j − 1] + gap_score)
21:        end for
22:    end for
23:    return score_opt(dpm[|Ĉ|])
24: end function
```

in the array at the position corresponding to the effect of the match on the two subsequences. The result for the current array element is the maximum result produced by any of these three cases. Once the array is filled, the result is the largest score anywhere in the row representing the end of the code snippet substring. The complexity of the algorithm is proportional to the size of the array *i.e.*, $O(|\widehat{C}| \times |\widehat{L}|)$.

4.2.4 Optimization

We then further optimize Algorithm 5 by reducing the number of columns in the matrix, representing the distance into the local function substring for which it is worth considering a match. Indeed, in the tail of $\widehat{L}$, the penalty of the leading gap alignment could eventually outweigh even a perfect alignment score for the remaining code snippet substring. Therefore, the optimal alignment can only involve instructions found within a certain distance R from the start of $\widehat{L}$. Let *max_score* be the maximal alignment value for two elements, *i.e.*, the score when the elements match perfectly, *gap_score* be the alignment value if either element is a gap, and *min_score* be the minimal alignment value, *i.e.*, the score when two elements are matched, but they are vastly different. Then, R is the smallest value such that $|\widehat{C}| \times min\_score > |\widehat{C}| \times max\_score + (R - |\widehat{C}|) \times gap\_score$, *i.e.*, $R > |\widehat{C}| \times ((min\_score - max\_score + gap\_score)/gap\_score)$, since *gap_score* is negative. This inequality says that a complete mismatch of each element of the code snippet substring at the

beginning of the local function substring could provide a higher alignment score than any perfect alignment that appears later in the sequence, beyond a sequence of gaps. Thus, we can just populate the matrix up to column R, instead of the full length $|\widehat{L}|$. The inequality also shows that R is of the same order of magnitude as $|\widehat{C}|$, as the least R satisfying the inequality is bounded by a constant multiple of $|\widehat{C}|$.

This optimization reduces the complexity of sequence alignment to $O(|\widehat{C}| \times R)$. Since (*min_score* − *max_score* + *gap_score*) / *gap_score* is a positive constant, the complexity becomes $O(|\widehat{C}|^2)$. Because the Linux kernel limits the length of the code snippet substring to a small constant (64 bytes), $O(|\widehat{C}|^2)$ amounts to a constant.

The anchored sequence matching algorithm uses the dynamic programming based sequence alignment algorithm twice per anchor point. As the complexity of the latter is constant, due to the restricted size of the code snippet, the overall complexity of Anchored Sequence Matching is linear in the number of anchor points.

4.3 Design Decisions

The matching algorithm is independent of how anchor points are selected and how the score associated with a match between two elements is determined. We now address these issues. We also consider the need to normalize the input in some cases and address the possibilities of ties.

4.3.1 Anchor Point Selection

The complexity of the Anchored Sequence Matching algorithm is proportional to the number of anchor points. The accuracy of the algorithm also depends heavily on the selection of anchor points. All suspicious points in the source sequence should be included as anchor points to ensure the soundness of the algorithm. However, we found that taking all points in the source sequence as possible anchor points hurts the accuracy, as some incorrect anchor points could obtain higher scores than the true counterpart point.

Our experiments showed that the counterpart instruction is often consistent with the trapping instruction in three ways: 1). The type of the opcode 2). The addressing modes of the operands 3). The offsets used in indirect addressing modes. For example, the instructions <mov 0x68(%rcx), %rcx> and <mov 0x68(%rax),%rax> have the same opcode type (mov), the same addressing mode (an indirect memory access for the source operand and a direct register access for the target operand), and the same offset of any indirect address (0x68), although they use different registers. Furthermore, we consider two instructions to be of the same type even if they operate on operands of different sizes (*e.g.* movl *vs.* movq) or their triggering conditions are opposite (*e.g.* je *vs.* jne).

We define a function *compare*(i_1, i_2) that compares two instructions i_1 and i_2 based on the above criteria. The result is a triple ⟨*type*, *mode*, *offset*⟩ where each element corresponds to a binary value that represents the comparison result with respect to he corresponding criterion,

e.g.,

$$compare(\frac{\text{<mov 0x68(\%rcx),\%rcx>}}{\text{<mov 0x68(\%rax),\%rax>}}) = \langle 1,1,1 \rangle$$

There are usually multiple anchor point candidates in the source sequence. To prioritize their selection, we build three queues based on the above criteria. The high-priority queue consists of the anchor points that satisfy all three criteria, *i.e.*, $\{i \mid compare(i,t) = \langle 1,1,1 \rangle\}$. The medium-priority queue consists of the ones that have the same instruction type as the trapping instruction, *i.e.*, $\{i \mid compare(i,t) = \langle 1,*,* \rangle\}$. And the low-priority queue consists of all the instructions in the local function. For the matching, we select the first non-empty queue following the order of the priority.

4.3.2 Scoring Function Design

The scoring function should measure the similarity of two related elements as well as the unlikelihood for the matching of two unrelated elements. The comparison between two elements can result in a *match*, a *mismatch*, or a *gap match* (*i.e.*, either element is empty). A *match* should have a higher score than a *mismatch*, and we previously argued that a gap match should have a negative score, to eliminate the match $(\perp,\perp)$ in any optimal solution [121]. We propose two possible scoring functions, differing in how much information about the instructions that they take into account.

Our first scoring function based on opcode, *score*, considers two instructions to be a *match* if they have the same type of opcode, *i.e.*, $compare(i_1,i_2) = \langle 1,*,* \rangle$, and any other comparison between two instructions to be a *mismatch*. We assign the scores 2 for a *match*, -1 for a *mismatch*, and -1 for a *gap match* [121]. In practice, we have observed that the *gap_match* value should not be lower than the *mismatch* value, or the algorithm will almost never assign a gap.

Our second scoring function based on both opcode and operands, $score'$, further refines the *match* case, following the comparison criteria proposed for the anchor point selection. For those instructions with both source and target operands, we give one point bonus if the addressing modes of the operands of the two instructions match, and another point if the offsets of indirect addressing modes match as well:

$$score'(i_1,i_2) = \begin{cases} 2 + mode + offset, & \text{if } type = 1 \\ score, & \text{otherwise} \end{cases} \quad \text{where } \langle type, mode, offset \rangle = compare(i_1,i_2)$$

4.3.3 Breaking Ties

Our algorithm selects anchor points and runs the sequence matching algorithm for each of them in order to identify the anchor point with the highest matching score. In case of a tie, we increase the score of each result with the best score by $score'(trap,anchor)$, thus doubling the weight of the anchor point. The strategy favours the anchor point that is more similar to the trapping instruction. If this computation again produces a tie, we then select the anchor point that occurs

Table 4.4: Distribution of trapping instructions

opcode	mov*	cmp*	test	add	and	ud2	inc	bts
count	73	14	5	2	2	2	1	1

earlier in the local function. This strategy is based on the observation that *e.g.*, if the oops is due to a dereference of a pointer that is NULL already at the beginning of the function, then only the first dereference attempt will be executed.

4.3.4 Input Normalization

The disassembled code snippet or local function may contain instructions such as <nop> (no operation), or <xchg %ax,%ax> (exchange the values of registers), that have no impact on the result of the execution, but that may *e.g.*, improve memory alignment. We refer to these instructions as *junk* instructions. The junk instructions might interfere with our matching, since they could unnecessarily incur the cost of a mismatch or a gap match if the compiler of the victim kernel and the compiler of the local function do not introduce them according to the same strategy. We thus remove all junk instructions before matching.

4.4 Evaluation

We have implemented our approach as a tool named OOPSA. [3] We now evaluate OOPSA, presenting our experimental data and settings, the performance benchmarking, and an analysis of the failure cases.

4.4.1 Experimental Data

We have selected 100 kernel oops samples, of which 90 come from the kernel oops repository [12] and 10 are from the attachment of bug reports in the Linux kernel bugzilla [69]. OOPSA requires that a kernel oops have the following properties: I). It should contain a code snippet that can be properly decoded. II). Gcc should be able to generate debugging information mapping between the source code and assembly code of the crashing function. Among the kernel oopses satisfying these properties, we randomly selected the 100 samples. The distribution of trapping instructions among these 100 kernel oopses is shown in Table 4.4, with 73 being variants of the *mov* instruction. Furthermore, in 95 out of 100 cases, the trapping instruction involves a memory-based operation. In 19 cases an operand is a constant, which can make the instruction more distinguishable. Finally, the number of instructions for the code snippet is between 10 and 27, with 18 on average.

To evaluate OOPSA, we establish the ground truth for each kernel oops of the 100 cases by manually inferring the offending line through the semantic matching (*cf.*, Section 4.1.2). The

[3] OOPSA rhymes with a French phrase “où ça?" (where is it?).

reason we chose 100 as the amount of sampling, is because the above analysis requires up to several hours per oops which limits the number of cases that we can consider. For some of the bugzilla samples, the offending line is given in the subsequent discussion.

4.4.2 Experimental Settings

Our experiments use a 64 bit machine(Debian 3.2.54-2), the easily accessible mainline source code from kernel.org corresponding to the version mentioned in the kernel oops, gcc version 4.7.2 (Debian 4.7.2-5), and a kernel configuration file generated by the standard command `make ARCH=i386|x86_64 defconfig`, specifying a 32 or 64-bit architecture, as indicated by the oops. Thus, our approach does not burden the developer with hunting down distribution-specific debugging kernels or recreating the exact build environment used to create the victim kernel. The default compiling configuration as stated previously allows us to generate the assembly code for almost all crashing functions in our samples. For those exceptional cases (7 out of 100), we then used the .config file of Debian 3.2.54-2 machine, compiled the source file directly and chose the default value for every prompt option, in order to generate the assembly code for the crashing function.

Our experiments measure both the accuracy and the running time of our solution. Accuracy is measured as the percentage of cases where the result is consistent with the established ground truth. Running time is measured as the average execution time per sample, over the 100 samples, including the time for parsing the input, the time for running the matching algorithm, and the time for generating the output. We measure the running time on a Mac mini with an Intel Core i5 2.5 GHz.

Our tool, OOPSA, is implemented in OCaml, consisting of the parsing of the kernel oops, the parsing of assembly code, and the matching algorithms. The implementation amounts to 1645 lines of OCaml code. This OCaml code is compiled into optimized native code for the evaluation.

4.4.3 Performance Benchmarking

Overall, OOPSA automatically identifies the correct offending line in 92 cases (92% accuracy), while the approach merely relying on the instruction pointer only identifies the correct offending line in 26 cases (26% accuracy). For these 26 cases, OOPSA works as well.

In Section 4.3, we proposed several designs concerning the anchor point selection, the scoring function, the matching score adjustment and the input normalization. We now evaluate the impact of these design decisions.

The algorithms for anchor point selection and for the scoring function affect the treatment of every oops, and thus we consider them together. Table 4.5 shows the accuracy and the running time for each possible combination. Using every instruction as an anchor point, combined with the scoring function that takes only the type of the opcode into account gives the worst accuracy (64%). Extending the scoring function to also take properties of the operands into account improves the

Table 4.5: Performance with core settings

Scoring Function	**Anchor Point Selection**	
	No	Yes
opcode based	64% / 450ms	86% / 20ms
opcode-operand based	72% / 838ms	90% / 33ms

Table 4.6: Performance with peripheral settings

Input Normalization	**Score Adjustment**	
	No	Yes
No	90% / 33ms	91% / 33ms
Yes	91% / 31ms	92% / 31ms

accuracy (72%), but at a heavy performance penalty (>80%) since it requires more sophisticated parsing for each instruction. On the other hand, just with the operand type based scoring function, simply applying anchor point selection gives a greater improvement in accuracy (86%), because it eliminates anchor points that mislead the algorithm, and also greatly improves the running time (22 times faster), because the complexity of the matching algorithm is linear in the number of anchor points. Further augmenting with the type-operand based scoring function, we obtain the best accuracy (90%). We furthermore note that the observed improvements in the number of correct results are monotonic; the number of correct results improves, while no previously correct results become incorrect.

Table 4.6 shows the impact of augmenting the use of the anchor point selection algorithm and the scoring function based on both the opcode and properties of the operands with the elimination of junk instructions and the treatment of ties. These optimizations, although not essential, help OOPSA deal with certain corner cases and have little impact on the running time.

For all the results shown in Tables 4.5 and 4.6, we have applied the optimization that reduces the size of matrix considered in the underlying dynamic programming algorithm, as presented in Section 4.2.3. The optimization greatly improves the running time of our solution, regardless of the other settings. Without this optimization, for example, the running times in the bottom row of Table 4.5 rise to 4253ms and 112ms respectively, an increase of up to over 5 times.

4.4.4 Failure Case Analysis

In six cases, either the local function is organized is a way that is profoundly different from the code snippet, so that the sequence matching algorithm is misled, or there are several very similar subsequences in the local function, and the sequence matching cannot distinguish among them. These cases break our assumption that the neighborhood of the counterpart instruction is more similar to that of the trapping instruction than that of any other instruction. For example, in one case, the crash occurs one line before a branching statement. In the code snippet, the branch is

implemented using the jne instruction, while in the local function, the branch is implemented using its opposite, je, changing drastically the instruction order. In another case, the oops is generated by a use of the BUG_ON macro, which occurs twice in the crashing function, in almost identical contexts. Both occurrences thus get the same score.

Second, our anchor point selection algorithm assumes that the opcode of the counterpart instruction is at least of the same type as the trapping instruction. In two cases, this assumption does not hold, leading to failure. For instance, in one case, the trapping instruction is a memory-access operation <movzwl 0x54(%rbx),%eax> followed by a test <test $0x2,%al>, while the counterpart instruction is a test <testb $0x2,0x54(%rbx)> combining both operations. As a result, the counterpart instruction is overlooked by the anchor point selection mechanism, since there are several candidates that match the trapping instruction better.

To address these cases, we plan to integrate more semantic analysis, such as data flow analysis, into the matching algorithm. One observation from the above example is that the <test $0x2,%al> instruction has a data dependency on the trapping instruction movzwl over the register %al. It should be noted that the test instruction is more identifiable, especially with a constant operand, since it appears less frequently than the variants of mov instruction. Thus, it may be more effective to find a counterpart instruction for the test instruction, instead for the trapping instruction, and to work back from there. We leave this as future work.

4.5 Threats to Validity

In this section, we discuss the threats to the various types of validity of our solution.

Construct Validity *Does the accuracy metric actually measure the accuracy of our solution?* The accuracy of the measurement depends on the ground truth. For each kernel oops sample, we establish the ground truth (its offending line) through both the assembly code matching and the semantic matching approaches. We found at least 3 pieces of evidence to support our assessment in each case. In addition, in certain samples from the Linux kernel bugzilla, the ground truth is given in the comments following the kernel oops, which reconfirms our assessment. There might, nevertheless, be some mistakes in our assessment.

Internal Validity *Are the underlying assumptions of our sequence matching algorithm valid in practice?* Our main assumptions are: I). If we find the counterpart instruction for the trapping instruction, then we can find the offending line. II). The counterpart instruction is situated in a neighborhood that is the most similar to the one of the trapping instruction. The first assumption depends on the functionality of gdb, which holds for all cases in our experiment. The second may not hold if the instruction sequence of the crashing function that led to the oops is significantly different from that of the mainline kernel. This can occur if the configuration options or source code are different; Linux is open source software, and anyone can freely recompile the kernel

under different configuration options and even modify the source code. Alternatively, as described in Section 4.4.4, there is a risk that the algorithm might be misled by multiple very similar regions of code within the crashing function.

External Validity *To which extent does the solution generalize to new kernel oopses, or to different platforms?* We argue that the samples in our experiments are representative, since they are extracted from well recognized sources: the kernel oops repository and the Linux kernel bugzilla. We are not aware of any other resources that provide more representative kernel oopses.

There are currently around 4400 kernel oopses with code snippets in the kernel oops repository. We obtained 92% accuracy on the 90 randomly selected samples from the repository. According to the sample size calculator,[4] our solution should work for at least 95%(-5.55) of all qualified kernel oopses. Therefore, we have a relatively high confidence that our solution works for any kernel oops that satisfies the properties listed in Section 4.4.

Furthermore, our sequence matching algorithm is primarily designed to locate a counterpart point in one sequence for a point of interest in another sequence. It is independent of the platform and the programming language. Therefore, it should be applicable to any scenario having a similar goal.

4.6 Conclusion

In this chapter, we have shown how to use approximate sequence matching to identify the offending line of a kernel oops. Our approach is fully automatic, from extracting the code snippet from the oops and obtaining the source code to returning the offending line. As such, our approach has great potential to help developers take advantage of the recently established kernel oops repository.

In future work, we will consider how to improve the accuracy of our approach, possibly by applying analyses such as dataflow analysis to the code snippet. We will also evaluate our approach on a larger set of oopses, and consider whether our approach could be relevant to other kinds of software.

[4] http://www.surveysystem.com/sscalc.htm

CHAPTER 5

RELATED WORK

Our anchored sequence alignment algorithm is designed to align a snippet of assembly code sequence to a longer assembly code sequence, given that the two sequences are generated from the same source code but with different compiling settings (compiler, compiling options, *etc*.). In another sense, our algorithm can also be used to measure the similarity of two assembly code sequences. Two similar assembly code sequences might be compiled from similar source code, or even the same source code but with different compiling settings as in our case. Based on this intuition, our algorithm could also be applied in another domain called ***code clone detection***. In the case of copy right violation, where a software product copies some parts of source code from another software product, our algorithm is more practical. Because the source code of software product is often not available, we then can only conduct code clone detection at the binary level (assembly code). Therefore, we review some related work in clone detection in this chapter.

5.1 Clone Detection

Code cloning [26, 114] occurs when fragments of code are copied and pasted with or without adaptation within or across software. It is a common phenomenon in software development. As shown in previous studies [16, 112], a significant fraction (between 7% and 23%) of code in a typical software system has been cloned. The practise of cloning is often intentional [72] and can be useful in some ways [13]. For example, to add a new feature into a software, a developer might prefer to reuse some existing code, as doing so is less likely to introduce errors than crafting completely new code. Meanwhile, it could be also harmful in software maintenance and evolution [65]. If the reused code is buggy, then the bug could proliferate through clones.

Code clone detection is essential for several purposes: I). ***refactoring***: in certain circumstance, fewer code clones means better code quality, while it is also considered to be a good practice to reuse some stable code instead of rewriting the code. A study conducted on a large telecommunication software system by Lagüe *et al.* [77] shows that a significant number of clones were removed due to refactoring, meanwhile the overall number of clones increased due to the faster rate of clone introduction. II). ***debugging***: if there is a bug in a code fragment, it is quite likely that its clones would have the same bug, or some other issues [41, 83]. III). ***protecting intellectual property***: reusing the source code from some third-party software without properly following its licence policy could cause some legal issues [53, 54].

In this section, we first define some basic concepts related to code clones. Then, we review some techniques and tools that perform code clone detection on the source code and the binary code respectively.

5.1.1 Definitions

In this section, we give a taxonomy that is commonly used by different techniques and tools related to clone detection.

Definition 5.1 (Code Fragment). *A code fragment (CF) is a sequence of code lines, including comments between lines [114]. A CF can be identified with three attributes: the source file where the CF resides, the beginning line of the CF within the file and the ending line of the CF. These are denoted as* $CF = (FileName, BeginLine, EndLine)$.

Definition 5.2 (Code Clone). *A code fragment* CF_1 *is a code clone of another code fragment* CF_2*, iff they are similar by certain measure of similarity which can be defined as a function in the form of* $Sim(CF_1, CF_2)$*. Two similar code fragments form a clone pair* (CF_1, CF_2).

There is not a universal definition of the similarity of code fragments. Other than 100% identical, some may claim that two code fragments that share more than 90% code of their code form a code clone, while others might argue that 80% is sufficient.

Definition 5.3 (Clone Class). *A set of code fragments, among which each pair is a clone pair, forms a clone class.*

Definition 5.4 (Clone Type). *Given a clone pair, one can classify it as 4 types (i.e. Type-1, Type-2, Type-3, Type-4), based on the degree of textual difference in the code.*

The categories range from a difference in layout or comments, which does not alter the syntactic or semantic meaning of the code lines, to a significant textual difference between two code fragments whose semantics are equivalent or at least approximately equivalent. We describe the clone types below in an incremental and recursive manner.

Type-1: Identical code fragments, except for variations in whitespace, layout and comments.

Type-2: Extends *Type-1* clones with additional variations in identifier names, literals or data types.

Type-3: Extends *Type-2* clones with further modifications such as added, changed, or removed statements.

Type-4: Clones with more significant syntactical variations than *Type-3*, that still share equivalent or similar semantics.

5.1.2 Source Code Clone Detection

Clone detection at the source code level has been well studied in the literature. Combining the classification of Roy *et al.* [114] and that of Jiang *et al.* [62], the techniques to detect source code clones can be grouped into four categories, based on the model used to represent source code, as shown in Table 5.1. We give a brief introduction for each category as follows:

Table 5.1: Classification [62, 114] of clone detection techniques at the *source code* level

techniques	references
string-based	**Dup** [15–17], Johnson [63], Marcus *et al.* [92], Roy *et al.* [113], Livieri *et al.* [89]
token-based	**CCFinder** [66], Gemini [124] CP-Miner [84], RTF [21], MOSS [118], JPLAG [105]
tree-based	**DECKARD** [62], CloneDR [22, 23], Wahler *et al.* [127], Mayrand *et al.* [93]
graph-based	**GPLAG** [87], Ferrante *et al.* [40], Weiser [130], Komondoor *et al.* [73, 74]

String-based: A program is parsed into a number of strings, usually divided by lines. A contiguous sequence of strings is grouped into a code segment. Two code segments are clones if their constituent strings match.

Token-based: A program is parsed into a number of tokens, *i.e.* the grammatical elements of a programming language, such as keywords, variable names and operators, *etc.*. Compared to the string-based approach, the token-based approach is usually more robust against code changes such as formatting and spacing.

Tree-based: A program is parsed into an abstract syntax tree (AST) where each node is a token within the program. The similarity of two code segments is then measured by matching the subtrees.

Graph-based: A program is parsed into a program dependence graph (PDG) [40], which combines the control flow graph (CFG) and the data dependency graph (DDG). Code clone detection is modelled as finding isomorphic subgraphs. This technique tends not to scale well. We give more details about PDG and the problem of graph isomorphism in the review of GPLAG [87] below.

Following the above classification, we select a tool (marked in **bold** in Table 5.1) from each category and elaborate the techniques that it uses.

Dup: A Program for Identifying Duplicated Code. Baker [15] developed a tool called **Dup**, that finds occurrences of duplicated or related code (clones) in large software systems to aid software maintenance and debugging. Baker chose a string-based (line-for-line) approach, based on the assumption that code cloning is accomplished by means of an editor, and thus the resulting clones will be largely the same line-for-line, or will be related in some systematic way such as variable renaming. The tool was applied to the source code of the X Window System and to part of a larger AT&T software system, and detected up to 38% of the entire code as being clones. However, the identified clones are less justifiable, especially for two cases: the initialization of large tables and the `switch-case` statements, where the code inherits a repetitive pattern but should not be considered as clones. Another shortcoming about the tool is that the code fragments identified in clones usually do not correspond exactly to subtrees in the AST tree of the program, or to any obvious semantic unit.

Dup works in two modes: exact matching and parameterized matching. Exact matching finds Type-1 clones, while parameterized matching finds Type-2 clones for which there is a one-to-one mapping between the corresponding variables in the clones. In exact matching, the tool first hashes each line of code to an integer and then constructs a sequence of integers from the input files. Then the tool uses a suffix-tree-based algorithm [94] to find the groups of longest common substrings within the generated sequence. Each group of common substrings corresponds to a code clone class. The suffix-tree-based algorithm is linear in the input, and a suffix tree can be built in time linear in the size of the input, for a fixed alphabet [94]. The parameterized matching is similar to the exact matching, except for two additional operations. The parameterized matching first replaces some tokens that could be candidates to be a parameter, then does an exact matching on the transformed code. Given the results produced by the exact matching, the parameterized matching filters out the ones that do not meet the one-to-one correspondence constraint between the parameters.

CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. Kamiya *et al.* [66] proposed a token-based clone detection algorithm and implemented it as a tool named **CCFinder**. CCFinder can detect clones within a single source file as well as across different files written in the same programming language. The authors applied the tool to several industrial and open-source projects in C, C++, Java, COBOL and a PL/I-like language. In addition, they claimed that the tool can be easily adapted to other programming languages. The tool can detect clones with different line breaks or different identifier names, *i.e.* Type-1 and Type-2 clones according to Definition 5.4. In a case study, they showed that CCFinder can find 23% more clones as compared to the string-based (line-by-line) tool Dup [15].

CCFinder performs clone detection in three main steps: I). ***Code Tokenization***: the input source files is parsed into tokens according to the lexical rules of the programming language. The tokens of all source files are then concatenated into a single token sequence, so that finding clones across multiple files is performed in the same way as finding clones in a single file. II). ***Token Normalization***: the parsed tokens are normalized based on some predefined rules, to tolerate the diverse syntactical formats and some slight variation of code between clones. For instance, the tool replaces each identifier related to a type, variable or constant with a special token, so that it can detect the Type-2 clones. III). ***Substring Extraction***: to the normalized token sequence, the tool applies the O(n) suffix-tree matching algorithm [49, 129] to extract lists of common substrings, which correspond to groups of code clones.

DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. Jiang *et al.* [62] proposed a novel algorithm for identifying similarity of subtrees and applied it to the AST of source code in order to detect code clones. More specifically, they proposed a representation model to capture the characteristics of a subtree with numerical vectors. Then, they measure the similarity of code fragments as the Euclidean distance between vectors. With the similarity function, they cluster the code fragments to identify the clone classes by using locality sensitive hashing (LSH) [36, 43]. LSH constructs a family of hashing functions that can hash two similar vectors to the same value with an arbitrary possibility as well as hashing two distant vectors to two remote values with an arbitrary possibility. This property of LSH allows the authors to avoid doing pairwise comparison which has a quadratic complexity and is expensive in practise. They integrated their algorithm into a tool named **DECKARD**. They conducted experiments on the Linux kernel and the JDK, to show that DECKARD is both scalable and accurate.

GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. To detect source code plagiarism, Liu *et al.* [87] proposed a PDG-based [40] plagiarism detection tool named **GPLAG**, which has been found to be more effective than the token-based algorithms such as MOSS [118] and JPLAG [105]. In order to make GPLAG scalable to large programs, the authors devised both lossless and lossy filters to prune the plagiarism search space. They conducted experiments on some UNIX command tools (*i.e. join, bc, less* and *tar*) to show that GPLAG is both effective and efficient.

GPLAG is built based on the observation that a plagiarised program is semantically equivalent to the original program. As the semantics of a program can be presented as a PDG, the plagiarism detection problem then reduces to isomorphism testing of PDGs. The definitions of some essential concepts that are involved in GPLAG are defined, as follows:

Definition 5.5 (**Program Dependence Graph**)**.** The program dependence graph G for a procedure P is a 4-tuple element $G = (V, E, \mu, \delta)$, where

- V is the set of program vertices in P, where each vertex could be a condition of control-flow, or a data manipulation operation such as a variable declaration, a value assignment, etc.
- $E \in V \times V$ is the set of dependency edges.
- $\mu : V \to S$ is a function assigning node types
  S = {ControlFlowNode, DataDependencyNode} to program vertices.
- $\delta : E \to T$ is a function assigning dependency types
  T = {ControlFlowEdge, DataDependencyEdge} to edges.

The size of a graph G is defined as the number of its vertices, *i.e.* $|G| = |V|$.

Definition 5.6 (Graph Isomorphism). A bijective function $f : V \to V'$ is a graph isomorphism from a graph $G = (V, E, \mu, \delta)$ to a graph $G' = (V', E', \mu', \delta')$ if

- the types of corresponding vertices match, *i.e.* $\mu(v) = \mu'(f(v))$
- there exists a corresponding edge between the corresponding vertices,
  i.e. $\forall e = (v_1, v_2) \in E \longrightarrow \exists e' = (f(v_1), f(v_2)) \in E' \quad \wedge \quad \delta(e) = \delta'(e')$
  and $\forall e' = (v'_1, v'_2) \in E' \longrightarrow \exists e = (f^{-1}(v_1), f^{-1}(v_2)) \in E \quad \wedge \quad \delta'(e') = \delta(e)$

Definition 5.7 (γ-Isomorphism). A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is isomorphic to G', and $|G'| = |S| \geq \gamma|G|, \gamma \in (0, 1]$.

The authors consider 5 common disguises that one may employ to plagiarise a program, as listed in Table 5.2. For each disguise, they argue that the essential part of the original PDG is unchanged, although the code appearance might be changed significantly. To compare two programs, GPLAG first parses them into two sets of PDGs, where each PDG corresponds to a procedure/function. Then GPLAG checks whether one function is plagiarised from another by checking whether their PDGs are isomorphic. For the latter check they use γ-isomorphism which is less strict than the standard isomorphism, in order to counter some tricker disguises that might alter the PDG of the original program. The definition of γ-isomorphism allows a graph g to be *isomorphic* to a smaller graph g', as long as the size of g' is not too small ($\geq \gamma|g'|$). This helps to detect the *code insertion* disguise. They set γ to 0.9 in their experiments, arguing that the effort of disguising more than 10% of a PDG is almost equivalent to rewriting the code, thus any effort beyond that threshold would contradict to the incentive of plagiarism which is to save effort.

Since subgraph isomorphism testing is in general a NP-complete problem, it could become intractable once the size of graph exceeds a certain limit. It is thus important to reduce the search space of the problem, which is the cartesian product ($|\varrho| * |\varrho'|$) of the sizes of the two PDG sets ϱ and ϱ' extracted from the programs to compare. The authors devised a lossless and a lossy filter to reduce the number of PDG pairs. The lossless filter is based on the definition of

Table 5.2: Common measures of plagiarism disguise

Measure	Description
Format alteration	Insert and remove blanks and/or comments.
Identifier renaming	Change the names of a function or its local variables.
Statement reordering	Reordering statements without altering data dependency.
Control replacement	Switch the `if{}` branches, replace the loop statement `for{}` with `while{}` or vice versa.
Code insertion	Insert some junk code that does not interfere with the logic of the original program.

γ-isomorphism. For two graphs g and g', if $|g'| < \gamma|g|$ then g cannot be γ-isomorphic to g'. The filter is lossless in the sense that no pair of γ-isomorphic PDGs is falsely excluded. The lossy filter is built on a similarity likelihood testing model named the G-test [106] that operates on the vertex histogram. This might generate some false negatives but is significantly cheaper to compute than isomorphism testing. The lossy filter allows users to tune the performance of GPLAG between efficiency and accuracy.

Code plagiarism is similar to clone cloning, in the sense that the correct results of code plagiarism detection are all qualified as code clones. However, they are different in several aspects: I). code cloning is a behaviour with several motivations of which plagiarism is only one. II). the semantics of plagiarised code is generally equivalent to that of the original code, while it is not necessarily true for a pair of code clones. For instance, one can copy an existing function and adapt it into a different usage scenario. III). the threshold of 10% PDG differences between the plagiarised code and its origin does not necessarily apply to code clones in general. For example, the adapted function in the previous example might be significantly different from its origin, in terms of the PDG.

5.1.3 Binary Code Clone Detection

In this section, we present a series of tools that can be used to detect code clones at the binary level. We summarize the techniques employed by these tools in Table 5.3.

Table 5.3: Clone detection techniques employed at the *binary* level

references	string literal	n-gram	CFG	sequence alignment	others
TRACY [37], PLDI'14			X	X	k-tracelet, SAT
Rendezvous [71], MSR'13	X	X	X		text indexing and querying (*CLucene*)
BAT [54], MSR'11	X				data compression, binary delta comparison
Sæbjørnsen *et al.* [116], ISSTA'09		X			clustering, LSH (Locality-sensitive Hashing)

TRACY: Tracelet-based code search in executables. David and Yahav [37] proposed a tracelet-based clone detection algorithm at the binary level. A *tracelet* is a sequence of instructions

that are bounded by control-flow instructions, *i.e.*, a basic block in the control-flow graph (CFG) parsed from the instruction sequence of a function. As the first part of their algorithm, the authors enumerate a *k-tracelet* set from the CFG, where each k-tracelet is composed of k directly-linked tracelets. To determine whether two functions are clones, the authors propose to compare the k-tracelet elements pairwise between the two functions. If the percentage of matched k-tracelets exceeds a threshold α, the functions are considered to match to each other. They calculate the similarity of two tracelets as the edit distance of the sequences of instructions involved in the tracelets. To improve the matching result, they first globally align the two instruction sequences composed from k-tracelets, and then use a constraint solver (SAT/SMT) to rewrite the aligned sequences to neutralise certain non-deterministic effects of compiler, such as assigning different registers for different optimisation levels, and finally recalculate the similarity of the rewritten sequences.

The authors did not provide an analysis of the complexity of their algorithm. But it is likely to be expensive. First, the algorithm involves a cartesian product between two sets of k-tracelets, where the size of each set is roughly proportional to the size of the corresponding function. Additionally, it invokes a constraint solving for each comparison, which is shown to take half of the time of the comparison. As a future work, the authors proposed to skip the constraint solving step in certain cases.

Rendezvous: A search engine for binary code. Khoo *et al.* [71] proposed a search engine called *Rendezvous* that enables indexing and searching for code in binary form. They abstract the assembly code with a statistical model as used in the domain of information retrieval (IR). Their model involves breaking up code into *tokens* and assigning a probability to the occurrence of each token in a reference corpus. They used the model to index a repository of binary code and also to construct a query string for a given binary in order to search for its clones in the repository. More specifically, they employed the schemes listed in Table 5.4 to index the binary code. We give a list of definitions for the concepts involved in this search engine as follows:

Table 5.4: Information retrieval schemes for assembly code

scheme	example	references
instruction mnemonic n-gram	[mov,sub] [sub,mov]	[99]
instruction mnemonic n-perm	{ mov, sub }	[67]
control flow sub-graph (k-graph)	[...jmp] ... → [...je]	[75]
extended control flow sub-graph	→ [...jmp] ... → [...je] →	[71]
data constant	$0x$BEEF	[54, 71]

Definition 5.8 (instruction mnemonic). *An instruction mnemonic is a textual description of the hexadecimal encoding of an operation code (opcode) in an instruction.*

The instruction mnemonic is architecture and platform dependent. Different processors have a different set of mnemonics. The same mnemonic might correspond to different opcodes: for instance, the opcodes 0x8b and 0x89 have the same mnemonic, mov.

Definition 5.9 (data constant). *Within the assembly code, a data constant is of three types: an integer operand, a NULL-terminated string operand, or an offset for memory addressing. For example, in the instruction <mov 0x4(%eax), 0x1>, there are two data contants: 0x4 and 0x1.*

Definition 5.10 (n-gram). *A sequence consisting of n tokens,* e.g. $[t_1, t_2, t_3]$. *The uniqueness and the order of the tokens involved in a n-gram distinguish it from other n-grams,* i.e. $[t_1, t_2, t_3] \neq [t_2, t_3, t_1]$.

Definition 5.11 (n-perm). *A set consisting of n tokens,* e.g. $\{t_1, t_2, t_3\}$. *Permutating the tokens within a n-perm does not generate a new n-perm,* i.e. $\{t_1, t_2, t_3\} = \{t_2, t_3, t_1\}$

Definition 5.12 (k-graph). *A sub-graph consisting of k nodes where each node is connected by at least one edge to at least one other node,* i.e. $(\{v_1, v_2, ...v_k\}, \{e_1, e_2, ...\})$.

Definition 5.13 (extended k-graph). *A k-graph extended with a virtual node* $\mathbf{v}^*$ *that represents the complete set of external nodes that have connections* $(e_1^*, e_2^*...)$ *to the internal nodes* $(v_1, v_2, ...v_k)$ *of the k-graph,* i.e. $(\{v_1, v_2, ...v_k, \mathbf{v}^*\}, \{e_1, e_2, ..., e_1^*, e_2^*...\})$

Definitions from 5.10 to 5.13 are schemes that can be used to capture various semantic or structural aspects of binary code. The instruction mnemonic *n-gram* and *n-perm* schemes partially capture the semantics of binary code. The n-gram enforces the order of terms (instructions) as well as the content. However, a slight reordering of the sequence of instructions could have no effect on the execution output. To capture this property, one can use the n-perm scheme by removing the order constraint of the n-gram. The k-graph scheme and its extended form capture the control-flow structure of the binary code. When the value of k is small (*e.g.* 3), there could be a lot of duplication among the k-graphs, while the extended k-graph can help to distinguish identical k-graphs in this case. And finally the *data constant* is one of the invariant features of binary code, based on the observation that data constants typically do not change with the compiler or compiler optimisation.

To index a given binary code, the Rendezvous engine extracts terms using all of the above schemes (*i.e.* n-gram, n-perm, k-graph and extended k-graph) and composes them together to construct a *signature* for the code. The signature is also used as a query to search for code clones in a reference corpus. The authors conducted experiments on functions from the GNU C library (*glib*) and the *coreutils* library. The functions are compiled both with the compiler *gcc* and *clang* under various levels of optimisation. The results of their experiments show that Rendezvous is able to retrieve cloned functions across binaries generated from different compilers and different optimisation levels, with a high efficiency and recall. Nevertheless, there is a threat to the

internal validity of their experiments, which is that it is possible that the *gcc* and *clang* compilers naturally produce similar binary code, at least for the software in the repository, and likewise, the results of different levels of compiler optimisation may be similar, at least with the `-O1` and `-O2` options of *gcc* in the experiments.

BAT: Binary Analysis Tool. To detect violations of the GNU General Public Licence, Hemel *et al.* [54] developed a tool called BAT (Binary Analysis Tool) for code clone detection in binaries. Given a binary, such as a firmware image, archive or compressed file (*i.e.* RAR, ZIP, tar, *etc.*), BAT detects clones from repositories of packages in source or binary form. Their definition of code clone includes the result of "copy and paste" cloning at the source code level, as well as the inclusion of third-party packages into the final binary through component composition methods such as static linking. BAT is thus designed to detect code clones at a coarse-grain level, *i.e.* API library, source code files, etc., instead of function clones. BAT comprises three clone detection techniques: scanning for string literals, detecting similarity through data compression, and detecting similarity by computing binary deltas.

A binary executable contains some string literals, such as constant strings in print statements, copyright strings, license strings and so on, that are unlikely to change among code clones [53]. The intuition of the string-literal based approach is that, if one finds that a large number of string literals extracted from a given binary also occur in another binary *foo*, it is reasonable to conclude that the binary contains a clone of some code in *foo*. As the requisite of BAT, they first built a string database that contains the string literals extracted from the source code files within a repository of open source packages. They then used the Unix *sed* script to strip comments from the source files and a regular expression to extract the remaining string literals. For the binary input, they used the GNU *strings* tool to extract the string literals. Finally, they designed a method to rank the list of packages that might be contained as clones in the given binary.

The main idea of the data-compression based clone detection technique is that if the concatenation of the subject binary and a binary from a precompiled open-source package has a smaller compression size than the sum of the sizes resulting from compressing each of them individually, then this is an evidence of redundancy between the two, and therefore cloning. They devised a metric to measure the "improvement" of compression if a binary contains a clone of a package from the repository. A crucial factor to make the technique work is the selection of the compression algorithm. In practice, the selected compression algorithm should be able to have a sliding window large enough to hold both files at the same time. Otherwise, the compression might not be able to fully exploit redundancy between the two files.

Finally, the binary-delta based clone detection technique is essentially a variant of the data compression method. The main idea is that, for a package x from the pre-compiled software repository, if the delta (*diff*) to reconstruct x from a given binary y is sufficiently small, which says a large part of x appear in y, then it indicates cloning.

The limitations of these approaches are that they all require a repository of pre-compiled binaries to check against and their results are architecture dependent. In addition, the data compression and delta comparison techniques are computationally intensive.

Detecting Code Clones in Binary Executables. Sæbjørnsen *et al.* [116] designed a practical clone detection algorithm for binary executables, based on clustering of feature vectors. A feature vector is a vector that contains a list of values, where each value is a digital representation of certain feature, *e.g.* count of instructions in a sequence. They represent the assembly code of a function as a list of feature vectors. Then they approximate the dissimilarity between two functions by the distance between vectors (such as Euclidean distance). They cluster the obtained feature vectors, where elements within the same clustering bucket are considered to be clones to each other.

To abstract binary code into feature vectors, they first extract a list of n-grams from a sequence of instructions. For each n-gram, they construct a feature vector, where each element is the occurrence count for a specific feature, such as the mnemonic `mov`, the type of operand (*i.e.* register, memory, constant), or the combination of the mnemonic and the type of the first operand in an instruction, *etc*. To reduce the space consumption as well as the computation time, they compressed the feature vectors using run-length encoding. Finally, they applied the locality-sensitive hashing (LSH) algorithm proposed in DECKARD [62] to do the clustering.

Miscellaneous. A code clone, by definition, is essentially two pieces of code that are similar to each other to certain degree. However, there does not seem to be a universal definition of whether two code snippets are clones of each other. For instance, one might claim 70% similarity is sufficient to be classified as code clone, while others might argument that 80% should be the threshold. Therefore, the validity of a code clone dataset is often under question. Recently, Krutz and Le [76] attempted to construct an validated code clone dataset as the test input for future research on clone detection. They ran multiple clone detection tools on a set of open source software projects and had the results manually cross-checked by experts.

5.2 Summary

In another sense, our anchored sequence alignment algorithm can also be used to measure the similarity between two assembly code sequences, since essentially the algorithm find the best matching point to align two sequences. Based on this intuition, our algorithm could also be applied in ***code clone detection*** at the binary code level. Therefore, in this chapter, we have also reviewed some related work in this domain, both on the *source code* and *binary code* levels. In order to detect a copy right violation, where a software product copies some parts of source code from another software product, code clone detection techniques at binary level are more practical, since the source code of commercial software products is often not available.

CHAPTER 6

CONCLUSION

To conclude, in this thesis, we have studied a specific kind of error report, *i.e.* a kernel oops from a real-world repository. Our work concerns two aspects of kernel oopses, *i.e. kernel oops comprehension* and *kernel oops debugging*. We detail our contributions respectively as follows:

Kernel oops comprehension. A kernel oops is a software artefact produced by the Linux kernel. A collection of kernel oopses potentially contains information that relates to the reliability of the Linux kernel. Therefore, we studied how we can interpret Linux kernel oopses to draw valid conclusions about Linux kernel reliability.

To this end, we have performed a study of over 187,000 Linux kernel oopses, collected in a repository maintained by Red Hat between September 2012 and April 2013. We first studied properties of the data itself, then correlated properties of the data with independently known information about the state of the Linux kernel, and then considered some features of the data that can be relevant to understanding the kinds of errors encountered by real users and how these features can be accurately interpreted. The main lessons learned are as follows:

- The number of oopses available for different versions varies widely, depending on which Linux distributions have adopted the version and the strategies used by the associated oops submission tools.
- The repository may furthermore suffer from duplicate and missing oopses, since the available information does not permit identifying either accurately.
- Identifying the service causing a kernel crash may require considering the call stack, to avoid merging oopses triggered in generic utility functions. The call stack, however, may

contain stale information, due to kernel compilation options.

- Analyses of the complete stack must take into account the fact that the kernel maintains multiple stacks, only one of which reflects the current process's or interrupt's execution history.

- Kernel executions may become tainted, indicating the presence of a suspicious action in the kernel's execution history. Oopses from tainted kernels may not reflect independent problems in the kernel code.

Kernel oops debugging. As a crash report, a kernel oops records the state information of the Linux kernel at runtime. Therefore, it serves as important information for kernel developers to debug an issue within the Linux kernel that triggered a kernel oops. We help the kernel developer to debug a kernel oops through pinpointing the offending line of the kernel oops.

To this end, we proposed a novel algorithm based on approximate sequence matching, as used in bioinformatics, to automatically pinpoint the offending line based on information about nearby machine-code instructions, as found in a kernel oops. We further implemented our approach as an automatic tool named **OOPSA** that takes only the kernel oops as input and produces the offending line number.

To evaluate OOPSA, we conducted a series of experiments on 100 randomly selected examples from the kernel oops repository and the Linux kernel bugzilla. OOPSA achieves 92% accuracy, compared to the 26% accuracy obtained through the state-of-the-art approach with the debugging tool GDB. In addition, our algorithm has complexity linear in the size of the crashing function. For the kernel oopses that we have tested, it takes on average 31ms to produce the result on a current commodity machine (2.5 GHz Intel Core i5). Therefore, our solution is effective and efficient, and thus it can ease the burden on a Linux kernel developer in debugging a kernel oops. In resolving this problem, we made the following contributions:

- We quantified the difficulty of finding the offending line in an oops generated by an older kernel.

- We proposed and formalized an algorithm using approximate sequence matching for identifying the line of source code that led to the generation of an oops.

- We showed that the complexity of our algorithm is linear in the size of the crashing function.

- We showed that our approach is effective, achieving an accuracy of 92% on 100 randomly selected examples.

6.1 Future Work

Expanding the survey of the kernel oopses. The study we conducted in Chapter 3 is based on the data collected between September 2012 and April 2013 in the kernel repository. The repository itself is constantly receiving kernel oops reports from Linux kernel deployments in real world. The lesson we learned from the experiments concern the properties of the content of kernel oopses, which are unlikely to change over time, *e.g.* the fact that the call stack in a kernel oops might be corrupt. However, the distribution of different kernel oopses varies over time, *e.g.* there might be more kernel oopses submitted from Debian than Fedora during certain time period. Thus, it would be interesting to re-conduct our experiments on a more recent data set from the kernel oops repository to have new results.

Moreover, as a more systematic approach, it might be worthwhile to build a set of scripts or tools that can be run periodically to re-perform the predefined experiments and to generate new results. The results could give developers more awareness on the current trend of kernel oopses so that they could prioritise their debugging effort, *i.e.* debugging the most frequently-reported kernel oopses. Indeed, as one can see from the current portal of the kernel oops repository `oops.kernel.org` [12], the site already shows rankings based on some attributes of kernel oops, *e.g.* the top guilty functions where the crashes occur. Some of our statistical experiments can complement the existing rankings and provide more perspectives on the kernel oops dataset. These experiments are good candidates to be fully automated and be re-run whenever possible.

Locating the NULL-pointer variable. Our tool OOPSA can help developers to narrow down the region of source code to search, from the hundreds of lines of code within the crashing function to a single offending line, which saves much human effort. However, the granularity of the result could be further improved. As we have observed from the results, the offending line is often a compound statement that may involve several pointer variables and the fault is often NULL-pointer dereference. In these cases, given the offending line, developers must still identify the exact variable that is the root cause of the NULL-pointer dereference. Therefore, it would be more helpful if the tool could automatically identify the NULL-pointer variable.

One of our directions of future work is to enhance our tool towards the above end. This could involve static code analysis on both the source code and the assembly code, which seems to be much more complicated than our current solution. However, it might not always be possible to obtain a result, due to the insufficient information. This enhancement is likely to be beneficial but remains as a challenge.

Program slice starting from the offending line. Our tool pinpoints the offending line, which is the last line of code that the Linux kernel ran before the kernel crash. The offending line is often the statement that involves *e.g.* a NULL-pointer dereference that triggers the crash. However, the offending line is not necessarily the root cause of the fault that leads to the crash,

but rather a starting point for developers to trace the cause. For example, the root cause of the crash might be some statement before the offending line, which accidentally frees the memory of the pointer variable that is dereferenced in the offending line. Thus, to find the root cause, developers should look into the code that may have impact on the offending line.

The statements that have impact on the offending line form a backward program slice [130]. The offending line together with all the pointer variables involved in the offending line can be considered as the slicing criterion. Within the program slice, there lies the root cause of the kernel crash. Therefore, another improvement that can be done given the offending line from our tool, is to perform backward program slicing starting from the offending line. The resulting program slice would save the debugging effort of developers, and help them to locate the root cause of the kernel oops more efficiently.

Many papers and tools provide algorithms for program slicing. However, effectively doing the program slicing of the offending line might also demand locating the NULL-pointer variable as explained in previous paragraph. Without discerning the actual NULL-pointer variable in the offending line, one would need to do multiple slices on each pointer variable involved in the offending line. Even so, overall, it would still save the effort of developers by giving all the possible program slices which would otherwise need to be derived by the developers manually.

Binary code clone detection. In this thesis, we align machine code sequences by comparing their similarity instruction-wise, using the semantics of assembly code, as well as sequence-wise, using the largest common subsequence of instruction sequences [100]. The algorithms and the methodologies that we devised are related to the ones used in the domain of binary code clone detection. We have reviewed the state-of-the-art on code clone detection in this thesis, to explore the possibility of applying our techniques to binary code clone detection. We are optimistic that our techniques might help to advance the state-of-the-art of binary code detection. Therefore, this remains a direction of future work.

Applicability to other platforms. In this thesis, we have targeted the Linux platform, and studied a specific kind of error report, *i.e.* a kernel oops, that is generated by the Linux kernel. Our work has been motivated and inspired by the study done on the error reports of the Windows platform [35, 44]. On the other hand, our work is distinguished from previous works in terms of the platform-specific usage scenarios and the algorithms we have designed. Our tool allows developers to pinpoint the offending line without instrumenting the system and without human intervention, which make it more suitable for postmortem debugging. Therefore, it would be interesting to see whether and how our work can be applied other platforms, *i.e.* Windows and Macintosh. This remains as a future work.

APPENDIX A

For those who prefer French, I present the Introduction, the Kernel Oops Debugging, and the Conclusion chapters of this thesis in French within this appendix.

A.1 Introduction

Dans ce chapitre, nous expliquons les problèmes que nous essayons de résoudre, et ensuite nous montrons les contributions que nous avons faites dans cette tentative. À la fin de ce chapitre, nous donnons une brève note sur le contenu de cette thèse.

A.1.1 Motivation

Le noyau Linux est aujourd'hui utilisé partout dans le monde numérique, allant des systèmes embarqués aux serveurs. Il réside au cœur de nombreuses distributions de systèmes d'exploitation, comme Android, Debian et Fedora. Bien que le noyau Linux soit largement considéré comme étant stable pour un usage ordinaire, néanmoins, il contient encore des bugs [103], *i.e.* des défauts dans le logiciel. Le noyau Linux est inévitablement bogué pour trois principales raisons. Tout d'abord, il dispose d'une énorme base de code de plus de 15 millions de lignes de code continuant de croître [33]. Il est extrêmement difficile de s'assurer de l'exactitude sur cette énorme base de code. Deuxièmement, le développement et la maintenance du noyau Linux sont très dynamiques. Il y a une version majeure du noyau Linux tous les 3 mois à laquelle plus d'un millier de développeurs du monde entier contribuent. L'introduction d'un nouveau code ainsi que les correctifs de bogues existants apporte de nouveaux bogues dans le noyau Linux. Troisièmement, le noyau Linux offre une grande variété de services (pilotes de périphériques, systèmes de fichiers, *etc.*) et chacun d'eux dispose d'une gamme d'options de configuration, ce qui entraîne un grand nombre de paramètres

de configuration pour le noyau Linux [119]. Par conséquent, il est pratiquement impossible de tester exhaustivement le noyau Linux pour toutes les configurations possibles. Pour ces raisons, il est impossible d'éliminer tous les bugs sur le noyau Linux. Les bugs persisteront tant que le noyau Linux évoluera mais à un taux tolérables grâce à un effort d'ingénierie continu.

Outre les mesures empêchant les bugs de se glisser dans le noyau Linux autant que possible, les développeurs du noyau Linux ont souvent recours au ***postmortem debugging*** (le débogage post-mortem). Le débogage est un processus permettant de localiser des bugs dans le logiciel, de trouver leurs origines et de fournir des correctifs. Le débogage post-mortem se réfère à l'approche de débogage d'un bug après sa manifestation, plutôt que de prédire un bug.

La manifestation d'un bug implique des *erreurs* qui sont des états inhabituels du logiciel. Comme un bug est souvent difficile à reproduire, il est essentiel d'intercepter et d'enregistrer les erreurs dues à celui-ci pour un débogage a posteriori. Afin de faciliter le débogage, les développeurs équipent couramment les logiciels avec une infrastructure de rapport d'erreurs, de sorte que le logiciel puisse générer automatiquement des informations sur les erreurs quand elles se produisent. Les rapports d'erreurs générés sont ensuite recueillis et servent plus tard de point de départ principal pour trouver des bugs.

Suivant cette même pratique, les développeurs du noyau Linux ont conçu celui-ci afin qu'il génére un type spécifique de rapport d'erreurs, appelé *kernel oops*, chaque fois que le noyau Linux plante. Entre 2007 et 2010, les kernel oopses ont tous été recueillis dans un dépôt sous la supervision de `kernel.org`, le site principal de la distribution du code source du noyau Linux. Il y avait une liste de diffusion [125] utilisée pour discuter et résoudre les kernel oopses les plus fréquents. De la même manière, l'habitude de maintenir et de déboguer des rapports d'erreurs a également été adoptée dans les systèmes Windows [97] et Mac OS X [1]. Les rapports d'erreurs du systéme Windows ont été utilisés pour des tâches telles que définir les priorités des efforts de débogage, détecter l'émergence des logiciels malveillants et suivre de la résurgence des bugs [44]. En effet, les données contenues dans ce dépôt ont le potentiel d'aider les développeurs et les chercheurs à identifier les problèmes importants dans l'implémentation du logiciel.

Dans cette thèse, nous étudions les rapports d'erreurs dans le contexte du noyau Linux, *i.e.* les kernel oops. Le caractère critique de la plupart des scénarios d'utilisation du noyau Linux signifie qu'il est important de comprendre et de résoudre rapidement les erreurs dans le noyau Linux qui sont rencontrées par des utilisateurs dans le monde réel. Nous abordons respectivement les deux objectifs ci-dessus, avec une étude systématique sur une grande collection de kernel oops et un outil pour aider les développeurs à mieux déboguer des kernel oopses.

A.1.2 Problème de Recherche

Nos objets d'étude sont les kernel oopses qui représentent un type particulier de rapports d'erreurs. Un kernel oops est émis par le noyau Linux lorsqu'un crash se produit. En partant d'un kernel oops, les développeurs du noyau Linux doivent comprendre l'information contenue dans

celui-ci puis trouver l'origine du problème qui l'a déclenché. Par conséquent, à propos des kernel oopses, notre problème de recherche porte sur deux aspects: ***la compréhension des kernel oops*** et ***le débogage des kernel oops***. En bref, la compréhension des kernel oops réside dans la compréhension de l'information portée par un kernel oops ainsi que la connaissance qui peut être extraite à partir d'une collection de kernel oops, tandis que le débogage de kernel oops se concentre sur les actions que l'on peut prendre en vue de trouver et de régler le problème qui déclenche le kernel oops. Nous discutons chaque aspect en détail dans les sections suivantes.

A.1.2.1 Compréhension de Kernel Oops

Un kernel oops contient des informations clés sur l'état du noyau Linux lors d'un crash. Un kernel oops est conçu pour être concis, contrairement à d'autres rapports d'erreurs similaires, par example, un kernel dump [111], qui est un instantané des données situées dans la mémoire et dont la taille varie de plusieurs centaines de Mo à plusieurs Go. La concision d'un kernel oops facilite son transfert mais inévitablement, il y a certaines informations qui ont été omises dans un kernel oops. Par conséquent, cela exige de la part des développeurs de récupérer et de dériver cette informations du contenu du kernel oops.

Les kernel oopses sont générés automatiquement dans le déploiement de Linux sur les sites des utilisateurs mais la récupération des kernel oopses est une action volontairede la part des utilisateurs. Depuis Septembre 2012, Red Hat a relancé la récupération de kernel oops dans le dépôt situé à `oops.kernel.org` [12] qui reçoit des rapports de kernel oops de toutes les distributions de Linux. Ce dépôt redonne alors la possibilité d'utiliser les kernel oops afin de comprendre les erreurs rencontrées par les utilisateurs du Linux. Néanmoins, il est difficile d'extraire des connaissances à partir d'une collection des kernel oopses. Le noyau Linux a des propriétés qui rendent l'interprétation des kernel oops difficile.

Le premier défi majeur dans l'interprétation du kernel oops est ***l'origine obscure***. L'une des informations les plus critiques dans un kernel oops est son origine, c'est à dire le code source qui génère le kernel oops et la plate-forme matérielle qui exécute le code source. Cependant, il est difficile, voire impossible, d'identifier avec précision l'origine d'un kernel oops. Le noyau Linux est fortement configurable, existe en de nombreuses versions et est intégré dans de nombreuses distributions comme Fedora, Debian, Android, *etc*. Dans un kernel oops, l'information de son origine est mal enregistrée. Même avec le numéro de version exact, il n'est pas facile d'associer un kernel oops avec son code source. Comme chaque distribution de Linux peut personnaliser le noyau Linux, il est difficile de suivre toutes les personnalisations. Quant à l'autre partie de renseignements sur l'origine, la plate-forme matérielle, elle n'est pas nécessairement présente dans un kernel oops. Le noyau Linux fonctionne sur un grand nombre de plates-formes matérielles, parmi lesquelles beaucoup sont propriétaires et dont les entreprises ne prennent pas la peine de remplir les informations de plate-forme dans le kernel oops. En effet, nous avons observé beaucoup de kernel oopses avec l'informations de plate-forme ayant pour dénomination `To be`

`filled` (à remplir). Nous soupçonnons que cela est en partie dû à des problèmes de confidentialité car de nombreuses entreprises ne sont pas disposées à révéler trop d'informations dans un rapport d'erreurs, même si elles acceptent de soumettre le rapport pour résoudre le problème. Cela peut être aussi simplement dû au fait que les administrateurs du système ne prennent pas la peine de remplir le champ.

Le deuxième défi majeur pour l'interprétation des kernel oopses est le problème de ***la forme incohérente***. Tout d'abord, les kernel oopses sont générés de manière ad-hoc, c'est-à-dire que la structure et le contenu d'un kernel oops dépendent du type d'erreur qui l'a déclenché. Par exemple, un déréférencement de pointeur non valide déclenche un kernel oops contenant un code machine extrait autour de l'adresse mémoire du pointeur d'instruction, mais ce fragment de code n'est pas présent dans les kernel oopses qui sont déclenchés par des erreurs liées au matériel. Deuxièmement, un kernel oops peut être transmis de manière différente selon la distribution de Linux utilisée. Pour des raisons de sécurité et de performance, le noyau Linux ne soumet pas de kernel oopses vers le dépôt mais les écrits dans un fichier de journalisation du noyau. Les différentes distributions de Linux fournissent leurs propres applications pour récupérer des kernel oopses de ce fichier de journalisation et les soumettre au dépôt. Ces applications pourraient décider de ne pas inclure certaines informations dans un kernel oops. Par exemple, nous avons remarqué que les versions récentes de Fedora suppriment l'identifier d'un kernel oops lors de la soumission.

En raison des défis décrits ci-dessus, il est difficile d'extraire toutes les informations nécessaires des kernel oopses et d'en tirer des conclusions fiables sur les informations. Dans le chapitre 3, nous effectuons une étude systématique sur de récents kernel oopses contenu dans un dépôt. En outre, nous détaillons les défis que nous avons rencontrés et les solutions que nous avons proposées.

A.1.2.2 Débogage de Kernel Oops

Une ensemble de kernel oops donne un aperçu sur le noyau Linux. D'un autre côté, un kernel oops individuel a le potentiel d'aider les développeurs du noyau Linux dans leur travail quotidien. Les principales tâches d'un développeur du noyau Linux sont d'ajouter de nouvelles fonctionnalités au noyau, ainsi que de résoudre les problèmes qui ont été identifiés dans l'implémentation des fonctionnalités existantes. Un kernel oops est utile pour cette dernière tâche. En effet, en tant que rapport d'erreur, un kernel oops enregistre une image instantanée de l'état du noyau au moment du crash. Par conséquent, un kernel oops fournit des informations de première main sur la manifestation d'un bug dans le noyau Linux, ce qui est essentiel pour résoudre ce bug plus tard. Dans le même temps, un kernel oops est enregistré principalement au niveau du code machine, ce qui le rend difficile à interpréter et limite considérablement son utilité. Ainsi, il est souhaitable pour les développeurs du noyau Linux d'avoir des techniques ou des outils qui peuvent les aider au débogage de kernel oops individuel.

Une partie fondamentales pour le débogage de tout crash de logiciel, y compris un kernel oops, est d'identifier l'*offending line* (la ligne fautive), c'est-à-dire la ligne du code source à laquelle le crash s'est produit. Afin d'illustrer l'importance de connaître la ligne fautive et la difficulté à l'obtenir à partir d'un kernel oops, même dans un cas favorable où il est possible d'interagir avec l'utilisateur qui a rencontré le crash, nous montrons un cas réel [28] extrait du bugzilla du noyau Linux. Un utilisateur nommé Jochen a rencontré des crashs fréquents sur son système de fichiers XFS et a déposé un rapport de bogue comprenant un kernel oops. Un développeur nommé Dave a pris en charge le rapport et a commencé à déboguer le problème. Comme Dave n'avait pas accès au noyau que Jochen avait utilisé, il a conduit Jochen au travers d'une série de procédures à effectuer sur sa machine afin de produire les informations à propos de la ligne fautive. Au total, il a fallu 16 échanges de commentaires avant qu'une autre utilisatrice, Katharine, qui a eu le même problème, ait réussi à obtenir la ligne fautive. Dave a rapidement compris la cause du bogue et a ensuite fourni un correctif correspondant.

Une solution pour obtenir les informations de ligne fautive serait d'instrumenter le noyau Linux pour inclure le numéro de la ligne fautive dans un kernel oops. Cependant, ce n'est pas une solution viable. Tout d'abord, la stratégie est intrusive car elle nécessite une mise-à-jour du noyau Linux. Comme des utilisateurs ont leur propre noyau Linux personnalisé, une mise-à-jour du noyau les obligerai à recompiler et recharger leur noyau. Cette exigence pourrait entraver l'adoption de la solution. Deuxiémement, l'instrumentation introduirait un surcoût d'exécution du noyau Linux, en plus d'augmenter la taille de l'image du noyau. En effet, une récente proposition (Septembre 2012) [70] pour étendre le noyau Linux afin d'inclure le numéro de la ligne du code source a été rejetée, car il exige un trop grand (bien que limité d'un certain point de vue) espace supplémentaire (10 Mo) dans le noyau. Pour finir et certainement le plus important, la solution ne résout pas le problème du débogage d'un grand nombre de rapports de kernel oops déjà disponible dans le rṕertoire, et qui possèdent un potentiel pour renforcer la fiabilité du noyau Linux s'il était possible de s'en servir.

Par conséquent, une solution souhaitable pour le problème de localisation précise de la ligne fautive d'un kernel oops serait d'avoir un outil automatique permettant de localiser celle-ci, en ayant uniquement un seul kernel oops, sans aucune intervention humaine ni instrumentation du système.

A.1.3 Contribution

Nous conduisons notre recherche au sujet de ces deux aspects de notre problème de recherche décrit ci-dessus. Dans cette section, nous listons les contributions que nous avons faites afin de résoudre le problème.

Compréhension de kernel oops. Nous étudions comment nous pouvons interpréter les kernel oopses pour en tirer des conclusions à propos de la fiabilité du noyau Linux. À cette fin,

nous effectuons une étude sur plus de 187 000 de kernel oopses, recueillis dans un dépôt maintenu par Red Hat entre Septembre 2012 et Avril 2013. Nous étudions d'abord les caractéristiques des données, puis nous les corrélons avec des informations connues indépendamment de l'état du noyau Linux. Pour finir, nous prenons en compte certaines caractéristiques des données qui peuvent être utiles pour comprendre les types d'erreurs rencontrées par les utilisateurs et comment ces caractéristiques peuvent être interprétées de façon précise. Les enseignements principaux que nous en avons tirés sont les suivants:

- Le nombre de kernel oopses disponibles dans le dépôt pour les versions différentes du noyau Linux est très variable. Cela dépend des outils de soumission pour les kernel oopses et de leurs stratégies sur chaque distribution de Linux.
- Le dépôt peut avoir des doublons ou des kernel oopses manquants car les informations disponibles ne permettent pas de vérifier l'un ou l'autre de façon précise.
- Identifier le service qui a déclenché un kernel oops peut nécessiter l'examen de la pile d'appel (call trace), afin d'éviter de fusionner les kernel oopses déclenchés dans des fonctions génériques.
- La pile d'appel, cependant, peut contenir des informations périmées, en raison des options de compilation.
- Les analyses de la pile complète d'appels de fonction doivent tenir compte du fait que le noyau maintient plusieurs piles, dont seulement l'une d'entre elles enregistre l'historique du processus courant ou de l'exécution de l'interruption.
- L'exécutions du noyau Linux peut être contaminé, indiquant la présence d'un action suspicieuse dans l'historique de l'exécution du noyau.
- Les kernel oops contaminés peuvent ne pas refléter les problémes indépendants dans le code du noyau Linux.

Débogage de kernel oops. Nous aidons les développeurs du noyau Linux à déboguer un kernel oops en localisant avec précision la ligne fautive du kernel oops. Pour cela, nous proposons un nouvel algorithme basé sur la correspondance approximative de séquences, utilisé en bio-informatique, afin d'identifier automatiquement la ligne fautive en se basant sur le code machine situé à proximité du point de crash que l'on trouve dans un kernel oops. Nous intégrons notre approche dans un outil automatique nommé **OOPSA** /où ça/, qui prend uniquement des kernel oopses en entrée et produit les numéros des lignes fautives.

Pour évaluer OOPSA, nous avons effectué un série d'expériences sur 100 exemples choisis au hasard dans le dépôt de kernel oopses et du bugzilla du noyau Linux. OOPSA obtient 92% de précision dans l'identification de la ligne fautive, par rapport à la précision de 26% obtenue par

l'approche de l'état de l'art actuel. De plus, notre algorithme une complexité linéaire en fonction de la taille de la fonction qui cause le crash. Pour les kernel oopses que nous avons testé, OOPSA prend en moyenne 31ms pour produire le résultat sur une machine actuelle. Par conséquent, notre solution est efficace et efficiente, elle peut donc alléger la charge d'un développeur du noyau Linux pour le débogage des kernel oopses.

En résolvant ce problème, nous avons fait les contributions suivantes:

- Nous quantifions la difficulté à trouver la ligne fautive dans un kernel oops généré par un noyau Linux plus ancien.
- Nous proposons et formalisons un algorithme qui utilise la correspondance approximative de séquences pour l'identification de la ligne du code source qui a déclenchée un kernel oops.
- Nous montrons que la complexité de notre algorithme est linéaire en fonction de la taille de la fonction qui cause le crash.
- Nous montrons que notre approche est efficace et atteint une précision de 92% sur 100 exemples choisis au hasard.

A.1.4 Organisation

Le reste de cette thèse est organisé comme suit: Le chapitre 2 présente l'état de l'art du domaine de cette thèse. Le chapitre 3 aborde l'aspect de la compréhension des kernel oopses qui viennent d'un dépôt maintenu par Red Hat. Le chapitre 4 discute de notre solution pour identifier la ligne fautive d'un kernel oops, dans le contexte du débogage de kernel oops. Le chapitre 5 discute des travaux qui sont liés aux techniques que nous avons conçu dans le Chapitre 4. Le chapitre 6 conclut cette thèse.

A.2 Débogage de Kernel Oops

Dans cette section, nous discutons de l'algorithme principal que nous avons conçu pour identifier la ligne fautive d'un kernel oops. L'objectif principal de l'algorithme est d'automatiser la mise en correspondance de l'extrait de code machine avec le code machine de la fonction compilée localement. Nous nous inspirons de *l'alignement approximatif des séquences* utilisé en bio-informatique [115, 121]. Nous commençons par quelques définitions et ensuite nous proposons un algorithme de correspondance qui se concentre sur chaque contrepartie possible à l'instruction fautive et à son contexte environnant. Cet algorithme de correspondance utilise un autre algorithme pour faire correspondre le préfixe d'une séquence avec une autre séquence de code, que nous présentons ci-après.

A.2.1 Définitions

Nous nous référons à l'extrait de code désassemblé comme étant le ***code extrait***, C, et à la fonction compilé localement comme étant la ***fonction locale***, L. Nous définissons d'abord quelques propriétés de séquence puis nous décrivons la correspondance de séquence:

Definition A.1. *Pour une séquence S, on note* $S[i]$ *l'élément à la position i et* $|S|$ *représente la longueur de S. Nous définissons également un élément spécial appelé* écart*, noté* $\perp$.

Definition A.2. *Pour une séquence S de longueur n et pour tout* (i,j) *tels que* $1 \leq i \leq j \leq n$, $\widehat{S} = S[i...j]$ *est une* **sous-chaîne** *de S et* $\widehat{S} = S[1...j]$ *est un* **préfixe** *de S.*

Definition A.3. *Pour les séquences* S_1 *et* S_2*, nous définissons un* ***alignement ancré*** *de* S_1 *avec* S_2*, noté* $\text{Align}(S_1,S_2)$*, comme n'importe quel couple* (S'_1,S'_2) *tel que* I). S'_1 *et* S'_2 *sont des séquences qui peuvent contenir des écarts.* II). $|S'_1| = |S'_2|$. III). *La suppression des écarts de* S'_1 *laisse* S_1. IV). *La suppression des écarts de* S'_2 *laisse un* préfixe *de* S_2.

L'alignement ancré est une variante de l'***alignement global*** et de l'***alignement local*** utilisé en bioinformatique [115, 121]. L'alignement global fait correspondre deux séquences complètes, tandis que l'alignement local fait correspondre des sous-séquences de ces séquences. l'alignement ancré fait correspondre une séquence complète à un préfixe d'une autre séquence, comprenant ou non des écarts. La Figure A.1 présente un exemple.

$S_1[1]$	$\perp$	$S_1[2]$	$S_1[3]$	$S_1[4]$	$\perp$	$\perp$	$S_1[5]$	
|	|	|	|	|	|	|	|	
$S_2[1]$	$S_2[2]$	$S_2[3]$	$S_2[4]$	$\perp$	$S_2[5]$	$S_2[6]$	$S_2[7]$	...

Figure A.1: $Align(S_1,S_2)$, où $|S_1| = 5$ et $|S_2| \geq 7$

En général, utiliser n'importe quelle correspondance pour trouver la ligne fautive ne suffit pas. À la place, nous choisissons la meilleure correspondance à l'aide d'une fonction de notation. En effet, le défi d'un alignement est de déterminer l'emplacement des écarts afin d'obtenir le score le plus élevé possible.

Definition A.4. *Soit les éléments* s_1 *et* s_2*, alors la fonction* **score**(s_1,s_2) *est une mesure de la similarité entre* s_1 *et* s_2.

La définition précise de la fonction *score* est orthogonale au problème de l'alignment et est reportée à la prochaine section. Néanmoins, nous requérons qu'une correspondance avec un écart ait un score négatif afin de le rendre défavorable lors d'une correspondance d'un écart avec lui-même.

Definition A.5. *Soit les séquences* S'_1 *et* S'_2 *de longueur n, pouvant contenir des écarts, alors nous définissons la* ***similarité*** *entre* S'_1 *et* S'_2*, comme* $\text{Sim}(S'_1,S'_2) = \sum_i^n \text{score}(S'_1[i],S'_2[i])$.

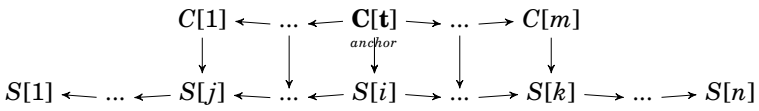

Figure A.2: Le schéma de l'alignment ancré des séquences

Definition A.6. *Soit les séquences S_1 et S_2, alors nous définissons θ comme l'ensemble de tous les alignements possibles ancrés entre S_1 et S_2, i.e.,* $\text{Align}(S_1, S_2)$*. Alors, l'**alignement ancré optimal** entre S_1 et S_2, noté* $\text{align}_{opt}(S_1, S_2)$*, est un élément de l'ensemble* $\{\theta \in \Theta \mid \forall \theta_1 \in \Theta, \text{Sim}(\theta) \geq \text{Sim}(\theta_1)\}$.

A.2.2 L'alignment Ancré des Séquences

Nous définissons la correspondance d'alignements ancrés de séquences comme un processus permettant de sélectionner *l'instruction homologue*, une instruction appartenant à la fonction locale et correspondant au mieux à l'instruction fautive, tout en prenant en compte le contexte de ces instructions. Tout d'abord, nous sélectionnons un ensemble de *points d'ancrage*, les candidats possibles appartenant à la fonction locale pour l'instruction homologue. Ensuite, à partir de chaque *point d'ancrage*, nous calculons respectivement la correspondance optimale entre les fragments précédents et suivants de l'extrait de code et de la fonction locale. Enfin, nous sélectionnons le point d'ancrage avec la valeur de similarité maximale pour l'instruction homologue.

L'Algorithme 6 présente l'algorithme de l'alignment ancré des séquences. Les lignes 2-6 effectuent les initialisations, y compris le fractionnement de l'extrait de code en trois parties: la partie précédent l'instruction fautive, *C_left*, l'instruction fautive, *trap*, et la partie suivant l'instruction fautive, *C_right*. Le séquence *C_left* est inversé car nous voulons faire correspondre la séquence à gauche de l'instruction fautive, précédent l'instruction fautive (*cf.* Figure A.2). Les lignes 8-22 itérent sur tous les points d'ancrage possibles. Pour chaque point d'ancrage, la fonction locale est divisée en *L_left*, *anchor*, et *L_right*, avec *L_left* inversé (lignes 9-12), de façon analogue à *C_left*. L'algorithme d'alignement ancré optimal est ensuite utilisé par la fonction **align** pour aligner *C_left* avec *L_left*, et *C_right* avec *L_right* pour obtenir le score de similarité (lignes 13-14). Nous calculons également le score de similarité de l'instruction fautive et du point d'ancrage choisi (ligne 15). Si la somme de ces scores est supérieure au meilleur score trouvé jusqu'à présent, le point d'ancrage actuel est enregistré comme étant la meilleure correspondance (lignes 16-21). Le résultat final est alors le meilleur point d'ancrage (ligne 23).

Algorithm 6 L'algorithme de l'alignment ancré des séquences

```
1: function ASM(List C, List L, List anchors, Int t)
2:     best_anchor ← −1
3:     max_score ← min_value
4:     C_left ← reverse(C[1..t − 1])
5:     trap ← C[t]
6:     C_right ← C[t + 1..C.length]
7:
8:     for i = 1 → anchors.length do
9:         L_left ← reverse(L[1..t − 1])
10:        anchor ← L[i]
11:        L_right ← L[t + 1..L.length]
12:
13:        left_score ← align(C_left, L_left)
14:        right_score ← align(C_right, L_right)
15:        anchor_score ← score(trap, anchor)
16:        total_score ← left_score + right_score + anchor_score
17:
18:        if (total_score > max_score) then
19:            max_score ← total_score
20:            best_anchor ← i
21:        end if
22:    end for
23:    return best_anchor
24: end function
```

A.2.3 L'alignment Ancré

L'alignement ancré des séquences utilise la fonction $align(\widehat{C},\widehat{L})$ pour faire correspondre l'extrait de code des chaînes de caractères $\widehat{C}$ précédent et suivant l'instruction fautive au contexte $\widehat{L}$ de chaque point d'ancrage proposé. L'Algorithme 7 présente une mise-en-œuvre récursive et simple. Pour chaque paire de séquences non-vides, cet algorithme prend la valeur maximale produite par deux possibilités: faire correspondre le début de chaque séquence avec entre eux et la fin de chaque séquence entre elles récursivement ou faire correspondre le début d'une séquence avec un écart et la fin de cette séquence récursivement avec l'autre séquence. Les lignes suivantes traitent les cas des séquences vides. Si la sous-chaîne d'extrait de code est vide (ligne 3), le résultat est 0, reflètant le fait que cela doit être simplement associé à un préfixe de la sous-chaîne de la fonction locale. D'autre part, si la sous-chaîne de la fonction locale est vide (ligne 4), le résultat est le produit de la longueur de la sous-chaîne de l'extrait de code restant et du score de l'écart, correspondant effectivement à la sous-chaîne de l'extrait de code restant avec les écarts.

L'algorithme récursif construit un arbre à trois voies d'exécution dans lequel chaque branche a une hauteur au plus de $|\widehat{C}|+|\widehat{L}|$. $|\widehat{C}|$ a une taille fixe, et sauf si la fonction locale est très petite, on a $|\widehat{C}| \leq |\widehat{L}|$. La complexité est par conséquent $O(3^{|\widehat{L}|})$.

L'algorithme récursif recalcule les scores de plusieurs alignements possibles. Pour éviter cette inefficacité, nous utilisons la technique de programmation dynamique [25] comme indiqué dans

Algorithm 7 Un algorithme récursif d'alignement ancré

```
1: function align_rc(List Ĉ, List L̂)
2:     match (Ĉ, L̂) with
3:     | ([ ], _) → 0
4:     | (_, [ ]) → |Ĉ| × gap_score
5:     | (c::ct, l::lt) →
6:       let gap_score  = score(⊥, ⊥) in
7:       let head_score = score(c, l)  in
8:       let no_gap = align(ct, lt) + head_score in
9:       let cs_gap = align(ct, L̂) + gap_score in
10:      let lk_gap = align(Ĉ, lt) + gap_score in
11:        return max(no_gap, cs_gap, lk_gap)
12: end function
```

l'algorithme 8. Cet algorithme remplit une matrice *DPM* de taille $(|\widehat{C}|+1)\times(|\widehat{L}|+1)$. La valeur stockée dans la cellule $dpm[i][j]$ représente le score de similarité du meilleur alignement entre $\widehat{C}[1..i]$ et $\widehat{L}[1..j]$. Comme l'alignement ancré nécessite de faire correspondre la sous-chaîne de l'extrait complet de code, mais seulement un préfixe de la chaîne de la fonction locale, nous prenons la meilleure valeur qui est trouvée n'importe où dans la ligne $dpm[|\widehat{C}|][*]$ comme le résultat. Ceci contraste avec l'implémentation utilisant la programmation dynamique de l'alignement global [115, 121], dans lequel le calcul itératif est le même, mais le résultat est $dpm[|\widehat{C}|][|\widehat{L}|]$.

Algorithm 8 L'alignement ancré par la programmation dynamique

```
1: function Align_dp(List Ĉ, List L̂)
2:     // init matrix with zero
3:     dpm ← make_matrix(0..|Ĉ|, 0..|L̂|, 0)
4:     gap_score ← score(⊥, ⊥)
5:
6:     for i = 1 → |Ĉ| do
7:         dpm[i][0] ← i * gap_score
8:     end for
9:
10:    for j = 1 → |L̂| do
11:        dpm[0][j] ← j * gap_score
12:    end for
13:
14:    for i = 1 → |Ĉ| do
15:        for j = 1 → |L̂| do
16:            head_score ← score(Ĉ[i], L̂[j])
17:            dpm[i][j] ← max (
18:                dpm[i − 1][j − 1] + head_score,
19:                dpm[i − 1][j] + gap_score,
20:                dpm[i][j − 1] + gap_score)
21:        end for
22:    end for
23:    return score_opt(dpm[|Ĉ|])
24: end function
```

L'algorithme utilisant la programmation dynamique (présenté dans l'Algorithme 8) commence

par l'initialisation de la colonne d'indice zéro de chaque ligne avec un multiple du score de l'écart (lignes 6-8), ce qui reflète la nécessité de faire correspondre des éléments de la sous-chaîne de l'extrait de code avec des écarts lorsque la sous-chaîne de la fonction locale est terminée. La ligne d'indice zéro de chaque est ensuite initialisée avec un multiple du score de l'écart (ligne 10-12), reflétant le coût de pousser l'alignement initial d'une sous-chaîne d'un extrait de code après un préfixe de la chaîne de la fonction locale, ce qui nécessite l'alignement des éléments de ce préfixe avec des écarts. Le reste de l'algorithme itére sur tous les éléments de la matrice, en partant des éléments ayant les plus petits indices (représentant les premières parties de chaque séquence) et se déplaçant vers les éléments ayant les plus grands indices (représentant les parties ultérieures de chaque séquence). Pour chaque paire de décalages, l'algorithme considère les possibilités que les éléments de la séquence représentées par la position courante correspondent (ligne 18), ou que l'un ou l'autre corresponde à un écart (lignes 19-20). Dans chaque cas, le score de correspondance ou le score d'écart est ajouté à la valeur calculée précédemment contenue dans la matrice à la position correspondant à l'effet de la correspondance sur les deux sous-séquences. Le résultat pour l'élément actuel de la matrice est le résultat maximal produit par l'un de ces trois cas. Une fois que le tableau est rempli, le résultat est le plus grand score situé dans la ligne, représentant la fin de la sous-chaîne d'extrait de code. La complexité de l'algorithme est proportionnelle à la taille de la matrice, *i.e.* $O(|\widehat{C}| \times |\widehat{L}|)$.

A.3 Conclusion

Dans cette thèse, nous avons étudié un type particulier de rapport d'erreurs, *i.e.* un kernel oops qui vient d'un dépôt du monde réel. Notre travail concerne deux aspects des kernel oopses: *la compréhension de kernel oops* et *le débogage de kernel oops*. Nous détaillons nos contributions comme suit:

Compréhension de kernel oops. Un kernel oops est un artefact de logiciel qui est généré par le noyau Linux. Une collection de kernel oopses contient potentiellement des informations liées à la fiabilité du noyau Linux. Par conséquent, nous avons étudié la façon dont nous pouvons interpréter les kernel oopses pour en tirer des conclusions valables sur la fiabilité du noyau Linux.

Nous avons effectué une étude sur plus de 187 000 kernel oopses qui ont été recueillis dans un dépôt maintenu par Red Hat entre Septembre 2012 et Avril 2013. Nous avons tout d'abord étudié les caractéristiques de ces données, puis nous avons corrélé les caractéristiques avec des informations connues indépendamment de l'état du noyau Linux. Nous avons pris en compte certaines caractéristiques de ces données pouvant être utiles pour comprendre les types d'erreurs rencontrées par les utilisateurs, et la façon précise dont ces caractéristiques peuvent être interprétées. Les enseignements principaux que nous avons tirés sont les suivants:

- Le nombre de kernel oopses disponibles dans le dépôt pour les versions différentes du noyau Linux est très variable Cela dépend des outils de soumission pour les kernel oopses et de leurs stratégies sur chaque distribution de Linux.
- Le dépôt peut avoir des doublons ou des kernel oopses manquants car les informations disponibles ne permettent pas de vérifier l'un ou l'autre de façon précise.
- Identifier le service qui a déclenché un kernel oops peut nécessiter l'examen de la pile d'appel (call trace), afin d'éviter de fusionner les kernel oopses déclenchés dans des fonctions génériques.
- La pile d'appel, cependant, peut contenir des informations périmées, en raison des options de compilation.
- Les analyses de la pile complète d'appels de fonction doivent tenir compte du fait que le noyau maintient plusieurs piles, dont seulement l'une d'entre elles enregistre l'historique du processus courant ou de l'exécution de l'interruption.
- L'exécutions du noyau Linux peut être contaminé, indiquant la présence d'un action suspicieuse dans l'historique de l'exécution du noyau.
- Les kernel oops contaminés peuvent ne pas refléter les problémes indépendants dans le code du noyau Linux.

Débogage de kernel oops. Comme un rapport d'erreur, un kernel oops enregistre des informations d'état du noyau Linux lors de l'exécution. Par conséquent, il contient des informations essentielles pour les développeurs de déboguer le probléme dans le noyau Linux qui a déclenché un kernel oops. Nous aidons le développeur du noyau Linux de déboguer un kernel oops par localiser avec précision la ligne fautive.

Nous avons proposé un nouvel algorithme basé sur l'algorithme de la correspondance approximatif de séquence, originé de la bio-informatique, afin d'identifier automatiquement la ligne fautive basé sur l'informations des codes matériels à proximité du point d'écrasement que l'on trouve dans un kernel oops. Nous avons intégré notre approche dans un outil automatique nommé **OOPSA** /où ça/, qui ne prend que des kernel oopses en entrée et produit les numéros des lignes fautives.

Pour évaluer OOPSA, nous avons effectué un série d'expériences sur 100 exemples choisis au hasard dans le dépôt du kernel oops et du bugzilla du noyau Linux. OOPSA a réalisé 92% de précision dans l'identification de la ligne fautive, par rapport à la précision de 26% obtenu par l'approche courante. De plus, notre algorithme a la complexité linéaire en fonction de la taille de la fonction qui s'écrase. Pour les kernel oopses que nous avons testé, il prend 31ms en moyens pour produire le résultat sur une machine actuel. Par conséquent, notre solution est efficace et

efficient, et donc il peut alléger la charge sur un développeur du noyau Linux pour la tâche de déboguer des kernel oopses.

En résoudrant ce problème, nous avons fait les contributions suivantes:

- Nous avons quantifié la difficulté à trouver la ligne fautive dans un kernel oops généré par un noyau Linux plus ancien.
- Nous avons proposé et formalisé un algorithme qui utilise la correspondance approximative de séquences pour l'identification de la ligne du code source qui a déclenchée un kernel oops.
- Nous avons montré que la complexité de notre algorithme est linéaire en fonction de la taille de la fonction qui cause le crash.
- Nous avons montré que notre approche est efficace et atteint une précision de 92% sur 100 exemples choisis au hasard.

A.3.1 Perspectives

Étendre l'étude des kernel oopses. L'étude que nous avons faite dans le Chapitre 3 est basé sur des données recueillies entre Septembre 2012 et Avril 2013 dans un dépôt de kernel oopses. Le dépôt reçoit constamment des rapports de kernel oopses à partir des déploiements du noyau Linux dans le monde réel. Les leçon que nous avons apprises concernant les caractéristiques du contenu d'un kernel oops sont peu susceptibles de changer au fil du temps. Par exemple, la pile d'appel dans un kernel oops court toujours le risque d'être contaminé. Cependant, la distribution de kernel oops des différents noyaux varie au cours du temps. Par exemple, il pourrait y avoir plus de kernel oopses de Debian par rapport á Fedora pendant certaines périodes de temps. Ainsi, il serait intéressant de re-conduire nos expériences sur des données plus récentes dans le dépôt de kernel oopses afin d'avoir des nouveaux résultats.

En outre, dans le cadre d'une approche systématique, il pourrait être intéressant de construire un ensemble de scripts ou d'outils pouvant être exécutés périodiquement et des expériences prédéfinies permettant de générer des nouveaux résultats. Les résultats pourraient donner aux développeurs une plus grande sensibilisation sur la tendance actuelle des kernel oopses afin qu'ils puissent établir une priorité dans leurs efforts de débogage, *i.e.* déboguer les kernel oopses les plus fréquents. En effet, comme on peut le voir sur le portail actuel du dépôt de kernel oopses (`oops.kernel.org` [12]), le site montre déjà des classements basés sur des attributs de kernel oops, par exemple, les fonctions qui font crasher le noyau le plus fréquemment. Une partie de nos expériences statistiques peuvent compléter les classements existants et offrir plus de perspectives sur l'ensemble des données sur les kernel oopses. Ces expériences sont de bons candidats pour être entièrement automatisées et être ré-exécutés à la demande.

Localisation d'un pointeur NULL. Notre outil (OOPSA) peut aider les dévelopeurs à identifer la région du code source à rechercher, partant de plusieurs centaines de lignes de code d'une fonction coupable à la ligne de code fautive, ce qui permet d'économiser des efforts d'ingénierie. Cependant, la granularité des résultats pourraient encore être améliorée. Comme nous l'avons observé dans les résultats obtenus, la ligne fautive est souvent une instruction composée qui implique plusieurs variables de pointeur, et la faute est souvent due au déréférence d'un pointeur NULL. Dans ces cas, compte tenu de la ligne fautive, les développeurs doivent encore identifier la variable exacte qui est la cause de racine de la déréférence d'un pointeur NULL. Par conséquent, l'outil serait plus utile s'il pouvait identifier plus précisement la variable de pointeur NULL.

L'une de nos perspectives de travail est d'améliorer notre outil afin de pouvoir remplir cet objectif. Cela pourrait employer de l'analyse statique de code, à la fois sur le code source (C) et le code assembleur, ce qui semble être beaucoup plus complexe que notre solution actuelle. Cependant, il ne serait pas toujours possible d'obtenir un résultat, notamment en raison de l'insuffisance des informations. Cette amélioration est susceptible d'être bénéfique, mais reste comme un défi.

Portion de programme partant de la ligne fautive. Notre outil identifie la ligne fautive qui est la dernière ligne de code que le noyau Linux exécute avant le crash du noyau. La ligne fautive est souvent une instruction qui implique, par exemple, le déréférencement d'un pointeur NULL qui provoque le crash. Cependant, la ligne fautive ne désigne pas nécessairement l'origine de la faute mais plutôt un point de départ pour les développeurs afin de la retrouver. Par exemple, l'origine de la faute pourrait être une instruction précédent la ligne fautive qui libère accidentellement une zone mémoire référencée par un pointeur et qui est ensuite accédée par la ligne fautive. Ainsi, pour trouver l'origine du problème, les développeurs devraient se pencher sur le code qui a un impact sur la ligne fautive.

Les états qui ont un impact sur la ligne fautive sont appelés "la portion arrière du programme" [130]. La ligne fautive avec l'ensemble des variables de pointeur impliqués dans la ligne fautive peuvent être considérés comme le critère de découpe du programme. Dans cette portion du programme réside la cause principale du noyau Linux. Par conséquent, compte tenu de la ligne fautive donnée par notre outil, une autre candidat d'amélioration est d'effectuer le tranchage de programme en arrière à partir de la ligne fautive. La portion du programme obtenue permettrait d'économiser du temps débogage pour les développeurs et de les aider à identifier les origines principales des kernel oopses plus efficacement.

De nombreux documents et outils fournissent des algorithmes pour une portion du programme. Cependant, en effectuer la découpe de programme en partant de la ligne fautive pourrait également nécessiter de localiser la variable de pointeur NULL comme expliqué dans le paragraphe précédent. Sans discerner la variable de pointeur NULL dans la ligne fautive, on aurait besoin de faire plusieurs découpes sur chaque variable pointeur impliquée dans la ligne fautive. Cela

étant dit, donner toutes les portions possibles du programme économiserait tout de même les efforts des développeurs au lieu que ce soit eux qui les extraient manuellement.

La détection de clone de code binaire. Dans cette thèse, nous alignons des séquences de code machine en comparant leur similitude instruction par instruction, en utilisant la sémantique du code assembleur, mais également séquence par séquence, en utilisant la plus grande sous-séquence commune de séquences d'instructions [100]. Les algorithmes et les méthodes que nous avons conçus sont aussi utilisés dans le domaine de la détection de clone de code binaire. Nous avons revu l'état de l'art des travaux sur la détection de clone de code dans cette thèse afin évaluer la possibilité d'appliquer nos techniques à ce domaine. Nous sommes confiants sur le fait que nos techniques peuvent contribuer à faire avancer l'état de l'art de la détection de clone de code binaire. Par conséquent, cela reste une perspective poru des travaux ultérieurs.

L'applicabilité à d'autres plates-formes. Dans cette thèse, nous avons ciblé la plate-forme Linux et étudié un type particulier de rapport d'erreurs *i.e.*, les kernel oopses qui sont générés par le noyau Linux. Notre travail a été motivé et inspiré par les études effectuées sur les rapports d'erreurs de la plate-forme Windows [35, 44]. D'autre part, notre travail se distingue des travaux antérieurs en terme de scénarios d'utilisation spécifiques à la plateforme et les algorithmes que nous avons conçus. Notre outil (OOPSA) permet aux développeurs d'identifier la ligne fautive sans instrumentation du système et sans intervention humaine, ce qui rend plus facile le débogage. Par conséquent, il serait intéressant de voir dans quelle mesure notre travail pourrait être appliqué sur d'autres plates-formes comme Windows et Macintosh.

BIBLIOGRAPHY

[1] *Apple Inc., CrashReporter. Technical Report TN2123, Cupertino, CA, USA, 2008.*

[2] *Automatic bug reporting tool (ABRT), Fedora.*

[3] *Kernel oops tracker (kerneloops), Ubuntu.*

[4] *Valgrind, an instrumentation framework for building dynamic analysis tools, valgrind.org.*

[5] R. ABREU, P. ZOETEWEIJ, AND A. J. C. V. GEMUND, *An evaluation of similarity coefficients for software fault localization*, in Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, PRDC '06, Riverside, CA, USA, Dec. 2006, IEEE Computer Society, pp. 39–46.

[6] ——, *Spectrum-based multiple fault localization*, in Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE, Auckland, New Zealand, Nov. 2009, IEEE Computer Society, pp. 88–99.

[7] R. ABREU, P. ZOETEWEIJ, AND A. J. C. VAN GEMUND, *On the accuracy of spectrum-based fault localization*, in Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, TAICPART-MUTATION '07, Washington, DC, USA, 2007, IEEE Computer Society, pp. 89–98.

[8] H. AGRAWAL, *On slicing programs with jump statements*, in Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, Orlando, Florida, USA, 1994, ACM, pp. 302–312.

[9] H. AGRAWAL, J. HORGAN, S. LONDON, AND W. WONG, *Fault localization using execution slices and dataflow tests*, in Proceedings of IEEE Software Reliability Engineering, Oct 1995, pp. 143–151.

[10] H. AGRAWAL AND J. R. HORGAN, *Dynamic program slicing*, in PLDI, White Plains, New York, USA, June 1990, pp. 246–256.

[11] S. F. ALTSCHUL, *A protein alignment scoring system sensitive at all evolutionary distances*, Journal of Molecular Evolution, 36 (1993), pp. 290–300.

[12] A. ARAPOV, *Kernel oops repository*, Sept. 2012.

[13] L. AVERSANO, L. CERULO, AND M. DI PENTA, *How clones are maintained: An empirical study*, in Proceedings of the 11th European Conference on Software Maintenance and Reengineering, CSMR, Amsterdam, the Netherlands, Mar. 2007, pp. 81–90.

[14] A. AVIZIENIS, J.-C. LAPRIE, B. RANDELL, AND C. LANDWEHR, *Basic concepts and taxonomy of dependable and secure computing*, IEEE Trans. Dependable Secur. Comput., 1 (2004), pp. 11–33.

[15] B. S. BAKER, *A program for identifying duplicated code*, in Computer Science and Statistics: Proc. Symp. on the Interface, March 1992, pp. 49–57.

[16] ——, *On finding duplication and near-duplication in large software systems*, in Proc. Working Conf. Reverse Engineering (WCRE), Los Alamitos, California, July 1995, pp. 86–95.

[17] ——, *Parameterized duplication in strings: Algorithms and an application to software maintenance*, SIAM J. Comput., 26 (1997), pp. 1343–1362.

[18] T. BALL AND S. G. EICK, *Software visualization in the large*, Computer, 29 (1996), pp. 33–43.

[19] T. BALL AND S. HORWITZ, *Slicing programs with arbitrary control flow*, in 1ST CONFERENCE ON AUTOMATED ALGORITHMIC DEBUGGING, Springer-Verlag, 1993, pp. 206–222.

[20] T. BALL AND J. R. LARUS, *Efficient path profiling*, in Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO, Paris, France, 1996, pp. 46–57.

[21] H. A. BASIT, S. J. PUGLISI, W. F. SMYTH, A. TURPIN, AND S. JARZABEK, *Efficient token based clone detection with flexible tokenization*, in The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, ESEC-FSE, Dubrovnik, Croatia, 2007, pp. 513–516.

[22] I. D. BAXTER, C. PIDGEON, AND M. MEHLICH, *DMS®: Program transformations for practical scalable software evolution*, in Proceedings of the 26th International Conference on Software Engineering, ICSE, Edinburgh, United Kingdom, May 2004, pp. 625–634.

[23] I. D. BAXTER, A. YAHIN, L. MOURA, M. SANT'ANNA, AND L. BIER, *Clone detection using abstract syntax trees*, in Proceedings of the International Conference on Software Maintenance, ICSM, Bethesda, Maryland, USA, 1998, pp. 368–.

[24] B. BEIZER, *Software Testing Techniques*, International Thomson Computer press, 2002.

[25] R. BELLMAN, *Dynamic Programming*, Princeton University Press, 1957.

[26] S. BELLON, R. KOSCHKE, G. ANTONIOL, J. KRINKE, AND E. MERLO, *Comparison and evaluation of clone detection tools*, Software Engineering, IEEE Transactions on, 33 (2007), pp. 577–591.

[27] J.-F. BERGERETTI AND B. A. CARRÉ, *Information-flow and data-flow analysis of while-programs*, ACM Trans. Program. Lang. Syst., 7 (1985), pp. 37–61.

[28] *Bug report in bugzilla: http://bugzilla.kernel.org/show_bug.cgi?id=27492*.

[29] C. CADAR, D. DUNBAR, AND D. 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, San Diego, California, 2008, pp. 209–224.

[30] M. Y. CHEN, E. KICIMAN, E. FRATKIN, A. FOX, AND E. BREWER, *Pinpoint: Problem determination in large, dynamic internet services*, in Proceedings of the 2002 International Conference on Dependable Systems and Networks, DSN, Bethesda, MD, USA, 2002, IEEE Computer Society, pp. 595–604.

[31] J.-D. CHOI AND J. FERRANTE, *Static slicing in the presence of goto statements*, ACM Trans. Program. Lang. Syst., 16 (1994), pp. 1097–1113.

[32] A. CHOU, J. YANG, B. CHELF, S. HALLEM, AND D. ENGLER, *An empirical study of operating systems errors*, in SOSP, 2001, pp. 73–88.

[33] J. CORBET, G. KROAH-HARTMAN, AND A. MCPHERSON, *Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*, The Linux Foundation, Mar. 2012.

[34] F. J. DAMERAU, *A technique for computer detection and correction of spelling errors*, Commun. ACM, 7 (1964), pp. 171–176.

[35] Y. DANG, R. WU, H. ZHANG, D. ZHANG, AND P. NOBEL, *ReBucket: a method for clustering duplicate crash reports based on call stack similarity*, in ICSE, Zurich, Switzerland, 2012, pp. 1084–1093.

[36] M. DATAR, N. IMMORLICA, P. INDYK, AND V. S. MIRROKNI, *Locality-sensitive hashing scheme based on p-stable distributions*, in Proceedings of the Twentieth Annual Symposium on Computational Geometry, SCG, Brooklyn, New York, USA, June 2004, pp. 253–262.

[37] Y. DAVID AND E. YAHAV, *Tracelet-based code search in executables*, in PLDI, Edinburgh, United Kingdom, June 2014, pp. 349–360.

[38] R. A. DEMILLO, H. PAN, AND E. H. SPAFFORD, *Failure and fault analysis for software debugging*, in Proceedings of the 21st International Computer Software and Applications Conference, COMPSAC, IEEE Computer Society, 1997, pp. 515–521.

[39] T. F. SMITH AND M. S. WATERMAN, *Identification of common molecular sequences*, Journal of Molecular Biology, 147 (1981), pp. 195–197.

[40] J. FERRANTE, K. J. OTTENSTEIN, AND J. D. WARREN, *The program dependence graph and its use in optimization*, ACM Trans. Program. Lang. Syst., 9 (1987), pp. 319–349.

[41] M. GABEL, J. YANG, Y. YU, M. GOLDSZMIDT, AND Z. SU, *Scalable and systematic detection of buggy inconsistencies in source code*, in Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA, Reno/Tahoe, Nevada, USA, 2010, pp. 175–190.

[42] V. GANESH AND D. L. DILL, *A decision procedure for bit-vectors and arrays*, in Proceedings of the 19th International Conference on Computer Aided Verification, CAV, Berlin, Germany, 2007, pp. 519–531.

[43] A. GIONIS, P. INDYK, AND R. MOTWANI, *Similarity search in high dimensions via hashing*, in Proceedings of the 25th International Conference on Very Large Data Bases, VLDB, Edinburgh, Scotland, UK, 1999, pp. 518–529.

[44] K. GLERUM, K. KINSHUMANN, S. GREENBERG, G. AUL, V. ORGOVAN, G. NICHOLS, D. GRANT, G. LOIHLE, AND G. HUNT, *Debugging in the (very) large: ten years of implementation and experience*, in SOSP, Big Sky, Montana, USA, 2009, pp. 103–116.

[45] T. GREGORY, *The economic impacts of inadequate infrastructure for software testing*, Tech. Rep. Planning Report 02-3.2002, National Institute of Standards and Technology, 2002.

[46] W. GU, Z. KALBARCZYK, K. RAVISHANKAR, AND Z. YANG, *Characterization of Linux kernel behavior under errors*, in DSN, June 2003, pp. 459–468.

[47] L. GUO, J. LAWALL, AND G. MULLER, *Oops! Where did that code snippet come from?*, in Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, Hyderabad, India, 2014, pp. 52–61.

[48] L. GUO, P. SENNA TSCHUDIN, K. KONO, G. MULLER, AND J. LAWALL, *Oops! what about a million kernel oopses?*, Tech. Rep. RT-0436, Inria, June 2013.

[49] D. GUSFIELD, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997, pp. 89–180.

[50] R. W. HAMMING, *Error detecting and error correcting codes*, Bell System Technical Journal, 29 (1950), pp. 147–160.

[51] S. HAN, Y. DANG, S. GE, D. ZHANG, AND T. XIE, *Performance debugging in the large via mining millions of stack traces*, in ICSE, Zurich, Switzerland, 2012, pp. 145–155.

[52] M. J. HARROLD, G. ROTHERMEL, R. WU, AND L. YI, *An empirical investigation of program spectra*, in Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '98, Montreal, Quebec, Canada, 1998, ACM, pp. 83–90.

[53] A. HEMEL, *The GPL compliance engineering guide. http://www.loohuis-consulting.nl/downloads/compliance-manual.pdf*, 2008-2012.

[54] A. HEMEL, K. T. KALLEBERG, R. VERMAAS, AND E. DOLSTRA, *Finding software license violations through binary code clone detection*, in MSR, Waikiki, Honolulu, HI, USA, May 2011, pp. 63–72.

[55] D. S. HIRSCHBERG, *A linear space algorithm for computing maximal common subsequences*, Commun. ACM, 18 (1975), pp. 341–343.

[56] S. HORWITZ, T. REPS, AND D. BINKLEY, *Interprocedural slicing using dependence graphs*, in Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI, Atlanta, Georgia, USA, 1988, ACM, pp. 35–46.

[57] D. R. HOWER AND M. D. HILL, *Rerun: Exploiting episodes for lightweight memory race recording*, in Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA, Washington, DC, USA, 2008, pp. 265–276.

[58] J. HUANG, P. LIU, AND C. ZHANG, *LEAP: lightweight deterministic multi-processor replay of concurrent java programs*, in SIGSOFT FSE, 2010, pp. 207–216.

[59] J. HUANG, C. ZHANG, AND J. DOLBY, *CLAP: Recording local executions to reproduce concurrency failures*, in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Seattle, Washington, USA, 2013, pp. 141–152.

[60] M. HUTCHINS, H. FOSTER, T. GORADIA, AND T. OSTRAND, *Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria*, in Proceedings of the 16th International Conference on Software Engineering, ICSE, Sorrento, Italy, 1994, IEEE Computer Society Press, pp. 191–200.

[61] D. JEFFREY, N. GUPTA, AND R. GUPTA, *Fault localization using value replacement*, in Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA, Seattle, WA, USA, 2008, ACM, pp. 167–178.

[62] L. JIANG, G. MISHERGHI, Z. SU, AND S. GLONDU, *DECKARD: Scalable and accurate tree-based detection of code clones*, in Proceedings of the 29th International Conference on Software Engineering, ICSE, Minneapolis, MN, USA, 2007, pp. 96–105.

[63] J. H. JOHNSON, *Substring matching for clone detection and change tracking*, in Proceedings of the International Conference on Software Maintenance, ICSM, Victoria, BC, Canada, 1994, pp. 120–126.

[64] J. A. JONES, M. J. HARROLD, AND J. STASKO, *Visualization of test information to assist fault localization*, in Proceedings of the 24th International Conference on Software Engineering, ICSE, Orlando, Florida, 2002, ACM, pp. 467–477.

[65] E. JUERGENS, F. DEISSENBOECK, B. HUMMEL, AND S. WAGNER, *Do code clones matter?*, in Proceedings of the 31st International Conference on Software Engineering, ICSE, Vancouver, British Columbia, Canada, 2009, pp. 485–495.

[66] T. KAMIYA, S. KUSUMOTO, AND K. INOUE, *CCFinder: A multilinguistic token-based code clone detection system for large scale source code*, IEEE Trans. Softw. Eng., 28 (2002), pp. 654–670.

[67] M. E. KARIM, A. WALENSTEIN, A. LAKHOTIA, AND L. PARIDA, *Malware phylogeny generation using permutations of code*, JOURNAL IN COMPUTER VIROLOGY, 1 (2005), pp. 13–23.

[68] S. KARLIN AND S. F. ALTSCHUL, *Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes*, in Proceedings of the National Academy of Science, vol. 87, USA, Mar. 1990, pp. 2264–2268.

[69] *Kernel bug tracker: https://bugzilla.kernel.org/*.

[70] M. KERRISK, *Kernel submit: Improving tracing and debugging*, 2012. https://lwn.net/Articles/514898/.

[71] W. M. KHOO, A. MYCROFT, AND R. ANDERSON, *Rendezvous: A search engine for binary code*, in Proceedings of the 10th Working Conference on Mining Software Repositories, MSR, San Francisco, CA, USA, 2013, pp. 329–338.

[72] M. KIM, V. SAZAWAL, D. NOTKIN, AND G. MURPHY, *An empirical study of code clone genealogies*, in Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, Lisbon, Portugal, 2005, pp. 187–196.

[73] R. KOMONDOOR AND S. HORWITZ, *Semantics-preserving procedure extraction*, in Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Boston, MA, USA, 2000, pp. 155–169.

[74] ——, *Using slicing to identify duplication in source code*, in Proceedings of the 8th International Symposium on Static Analysis, SAS, Paris, France, 2001, pp. 40–56.

[75] C. KRUEGEL, E. KIRDA, D. MUTZ, W. ROBERTSON, AND G. VIGNA, *Polymorphic worm detection using structural information of executables*, in Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection, RAID, Seattle, WA, 2006, pp. 207–226.

[76] D. E. KRUTZ AND W. LE, *A code clone oracle*, in MSR, Hyderabad, India, May 2014, pp. 388–391.

[77] B. LAGÜE, D. PROULX, J. MAYRAND, E. M. MERLO, AND J. HUDEPOHL, *Assessing the benefits of incorporating function clone detection in a development process*, in Proceedings of the International Conference on Software Maintenance, ICSM, Bari, Italy, 1997, pp. 314–.

[78] C. LATTNER AND V. ADVE, *LLVM: A compilation framework for lifelong program analysis & transformation*, in Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO, Palo Alto, California, 2004, pp. 75–.

[79] D. LEE, P. M. CHEN, J. FLINN, AND S. NARAYANASAMY, *Chimera: hybrid program analysis for determinism*, in PLDI, 2012, pp. 463–474.

[80] D. LEE, M. SAID, S. NARAYANASAMY, AND Z. YANG, *Offline symbolic analysis to infer total store order*, in HPCA, 2011, pp. 357–358.

[81] D. LEE, M. SAID, S. NARAYANASAMY, Z. YANG, AND C. PEREIRA, *Offline symbolic analysis for multi-processor execution replay*, in MICRO, 2009, pp. 564–575.

[82] V. I. LEVENSHTEIN, *Binary codes capable of correcting deletions, insertions, and reversals*, in Soviet Physics Doklady, 1966, pp. 707–710.

[83] J. LI AND M. D. ERNST, *CBCD: Cloned buggy code detector*, in Proceedings of the 34th International Conference on Software Engineering, ICSE, Zurich, Switzerland, 2012, pp. 310–320.

[84] Z. LI, S. LU, S. MYAGMAR, AND Y. ZHOU, *CP-Miner: A tool for finding copy-paste and related bugs in operating system code*, in Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI, San Francisco, CA, 2004, pp. 20–20.

[85] B. LIBLIT, A. AIKEN, A. X. ZHENG, AND M. I. JORDAN, *Bug isolation via remote program sampling*, in Proceedings of the ACM SIGPLAN 2003 Conference on Programming

Language Design and Implementation, PLDI, San Diego, California, USA, 2003, ACM, pp. 141–154.

[86] B. LIBLIT, M. NAIK, A. X. ZHENG, A. AIKEN, AND M. I. JORDAN, *Scalable statistical bug isolation*, in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Chicago, IL, USA, 2005, ACM, pp. 15–26.

[87] C. LIU, C. CHEN, J. HAN, AND P. S. YU, *GPLAG: Detection of software plagiarism by program dependence graph analysis*, in Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD, Philadelphia, PA, USA, 2006, pp. 872–881.

[88] C. LIU, X. YAN, L. FEI, J. HAN, AND S. P. MIDKIFF, *Sober: Statistical model-based bug localization*, in Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, Lisbon, Portugal, 2005, ACM, pp. 286–295.

[89] S. LIVIERI, Y. HIGO, M. MATUSHITA, AND K. INOUE, *Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder*, in 29th International Conference on Software Engineering, ICSE., Minneapolis, MN, USA, May 2007, pp. 106–115.

[90] D. LO AND S.-C. KHOO, *SMArTIC: Towards building an accurate, robust and scalable specification miner*, in FSE, Portland, Oregon, USA, 2006, pp. 265–275.

[91] LUCIA, D. LO, L. JIANG, AND A. BUDI, *Comprehensive evaluation of association measures for fault localization*, in 2010 IEEE International Conference on Software Maintenance (ICSM), Sept 2010, pp. 1–10.

[92] A. MARCUS AND J. I. MALETIC, *Identification of high-level concept clones in source code*, in Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE, San Diego, USA, 2001, pp. 107–.

[93] J. MAYRAND, C. LEBLANC, AND E. MERLO, *Experiment on the automatic detection of function clones in a software system using metrics*, in Proceedings of the 1996 International Conference on Software Maintenance, ICSM, Monterey, CA, USA, 1996, pp. 244–.

[94] E. M. MCCREIGHT, *A space-economical suffix tree construction algorithm*, J. ACM, 23 (1976), pp. 262–272.

[95] A. A. D. S. MEYER, A. A. F. GARCIA, A. P. D. SOUZA, AND C. A. L. D. SOUZA JR., *Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L)*, Genetics and Molecular Biology, 27 (2004), pp. 83 – 91.

[96] P. MONTESINOS, L. CEZE, AND J. TORRELLAS, *DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently*, in Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA, Washington, DC, USA, 2008, pp. 289–300.

[97] *Microsoft MSDN. Windows error reporting.*

[98] G. MULLER, J. L. LAWALL, AND H. DUCHESNE, *A framework for simplifying the development of kernel schedulers: Design and performance evaluation*, in HASE, Heidelberg, Germany, Oct. 2005, pp. 56–65.

[99] G. MYLES AND C. COLLBERG, *K-gram based software birthmarks*, in Proceedings of the 2005 ACM Symposium on Applied Computing, SAC, Santa Fe, New Mexico, 2005, pp. 314–318.

[100] S. B. NEEDLEMAN AND C. D. WUNSCH, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology, 48 (1970), pp. 443–453.

[101] K. J. OTTENSTEIN AND L. M. OTTENSTEIN, *The program dependence graph in a software development environment*, in Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SDE 1, ACM, 1984, pp. 177–184.

[102] Y. PADIOLEAU, J. LAWALL, R. R. HANSEN, AND G. MULLER, *Documenting and automating collateral evolutions in Linux device drivers*, in EuroSys, Glasgow, Scotland UK, Apr. 2008, pp. 247–260.

[103] N. PALIX, G. THOMAS, S. SAHA, C. CALVÈS, J. LAWALL, AND G. MULLER, *Faults in Linux: ten years later*, in ASPLOS, Mar. 2011, pp. 305–318.

[104] S. PARK, Y. ZHOU, W. XIONG, Z. YIN, R. KAUSHIK, K. H. LEE, AND S. LU, *PRES: probabilistic replay with execution sketching on multiprocessors*, in SOSP, 2009, pp. 177–192.

[105] L. PRECHELT, G. MALPOHL, AND M. PHLIPPSEN, *Finding plagiarisms among a set of programs with JPLAG*, J. of Universal Computer Science, (2002).

[106] R. R. SOKAL AND F. J. ROHLF, *Biometry: The Principles and Practices of Statistics in Biological Research*, W. H. Freeman; 3rd edition, 1994.

[107] M. K. RAMANATHAN, A. GRAMA, AND S. JAGANNATHAN, *Sieve: A tool for automatically detecting variations across program versions*, in ASE, 2006, pp. 241–252.

[108] M. K. RAMANATHAN, S. JAGANNATHAN, AND A. GRAMA, *Trace-based memory aliasing across program versions*, in FASE, 2006, pp. 381–395.

[109] T. REPS, T. BALL, M. DAS, AND J. LARUS, *The use of program profiling for software maintenance with applications to the year 2000 problem*, in Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC'97/FSE-5, Zurich, Switzerland, 1997, Springer-Verlag New York, Inc., pp. 432–449.

[110] T. REPS, S. HORWITZ, M. SAGIV, AND G. ROSAY, *Speeding up slicing*, in Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT, New Orleans, Louisiana, USA, 1994, ACM, pp. 11–20.

[111] J. RÖSSLER, A. ZELLER, G. FRASER, C. ZAMFIR, AND G. CANDEA, *Reconstructing core dumps*, in ICST '13: Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation, Luxembourg, Mar. 2013.

[112] C. K. ROY AND J. R. CORDY, *An empirical study of function clones in open source software*, in Proceedings of the 15th Working Conference on Reverse Engineering, WCRE, Washington, DC, USA, 2008, IEEE Computer Society, pp. 81–90.

[113] ——, *NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization*, in Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC, Amsterdam, the Netherlands, June 2008, pp. 172–181.

[114] C. K. ROY, J. R. CORDY, AND R. KOSCHKE, *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach*, Sci. Comput. Program., 74 (2009), pp. 470–495.

[115] W. L. RUZZO AND M. TOMPA, *A linear time algorithm for finding all maximal scoring subsequences*, in Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology, 1999, pp. 234–241.

[116] A. SÆBJØRNSEN, J. WILLCOCK, T. PANAS, D. QUINLAN, AND Z. SU, *Detecting code clones in binary executables*, in Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA, Chicago, IL, USA, 2009, pp. 117–128.

[117] S. SAHA, J.-P. LOZI, G. THOMAS, J. L. LAWALL, AND G. MULLER, *Hector: Detecting resource-release omission faults in error-handling code for systems software*, in DSN, Glasgow, Scotland UK, June 2013.

[118] S. SCHLEIMER, D. S. WILKERSON, AND A. AIKEN, *Winnowing: Local algorithms for document fingerprinting*, in Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD, San Diego, California, 2003, pp. 76–85.

[119] R. TARTLER, D. LOHMANN, J. SINCERO, AND W. SCHRÖDER-PREIKSCHAT, *Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem*, in EuroSys, 2011.

[120] F. TIP, *A survey of program slicing techniques*, Journal of programming languages, 3 (1995), pp. 121–189.

[121] M. TOMPA, *Lecture notes on biological sequence analysis*, Tech. Rep. 2000-06-01, Department of Computer Science and Engineering University of Washington, Winter 2000.

[122] E. TORLAK, M. VAZIRI, AND J. DOLBY, *MemSAT: Checking axiomatic specifications of memory models*, in Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Toronto, Ontario, Canada, 2010, pp. 341–350.

[123] L. TORVALDS, *http://www.kernel.org/doc/Documentation/oops-tracing.txt*.

[124] Y. UEDA, T. KAMIYA, S. KUSUMOTO, AND K. INOUE, *On detection of gapped code clones using gap locations*, in Software Engineering Conference, 2002. Ninth Asia-Pacific, 2002, pp. 327–336.

[125] A. VAN DE VEN, *Top 10 kernel oopses for the week ending january 5th, 2008. Mailing-list:*.

[126] K. VEERARAGHAVAN, D. LEE, B. WESTER, J. OUYANG, P. M. CHEN, J. FLINN, AND S. NARAYANASAMY, *DoublePlay: parallelizing sequential logging and replay*, in ASPLOS, 2011, pp. 15–26.

[127] V. WAHLER, D. SEIPEL, J. W. V. GUDENBERG, AND G. FISCHER, *Clone detection in source code by frequent itemset techniques*, in Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop, SCAM, Chicago, USA, 2004, IEEE Computer Society, pp. 128–135.

[128] D. WEERATUNGE, X. ZHANG, AND S. JAGANNATHAN, *Analyzing multicore dumps to facilitate concurrency bug reproduction*, in ASPLOS, 2010, pp. 155–166.

[129] P. WEINER, *Linear pattern matching algorithms*, in Proceedings of the 14th Annual Symposium on Switching and Automata Theory, SWAT, 1973, pp. 1–11.

[130] M. WEISER, *Program slicing*, in Proceedings of the 5th international conference on Software engineering, ICSE, San Diego, California, USA, Mar. 1981, pp. 439–449.

[131] W. E. WONG AND V. DEBROY, *A survey on software fault localization*, Tech. Rep. UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, Nov. 2009.

[132] T. YOSHIMURA, H. YAMADA, AND K. KONO, *Is Linux kernel oops useful or not?*, in HotDep, Hollywood, CA, USA, Oct. 2012.

[133] C. ZAMFIR AND G. CANDEA, *Execution synthesis: a technique for automated software debugging*, in Proceedings of the 5th European conference on Computer systems, EuroSys, Paris, France, 2010, pp. 321–334.

[134] X. ZHANG AND R. GUPTA, *Cost effective dynamic program slicing*, in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI, Washington DC, USA, 2004, ACM, pp. 94–106.

[135] X. ZHANG, R. GUPTA, AND Y. ZHANG, *Precise dynamic slicing algorithms*, in Proceedings of the 25th International Conference on Software Engineering, ICSE, Portland, Oregon, 2003, IEEE Computer Society, pp. 319–329.

[136] X. ZHANG, H. HE, N. GUPTA, AND R. GUPTA, *Experimental evaluation of using dynamic slices for fault location*, in Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG, Monterey, California, USA, 2005, ACM, pp. 33–42.

[137] J. ZHOU, X. XIAO, AND C. ZHANG, *Stride: Search-based deterministic replay in polynomial time via bounded linkage*, in Proceedings of the 34th International Conference on Software Engineering, ICSE, Zurich, Switzerland, 2012, pp. 892–902.