On the Efficient Design and Testing of Dependable Systems Software

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Vom Fachbereich Informatik
der Technischen Universität Darmstadt

genehmigte

Dissertation

zur Erlangung des akademischen Grades eines
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt von

Oliver Schwahn, M.Sc.

aus Heppenheim an der Bergstraße

Referenten:
Prof. Neeraj Suri, Ph.D.
Prof. Karthik Pattabiraman, Ph.D.

Tag der Einreichung: 15. Februar 2019
Tag der mündlichen Prüfung: 29. März 2019

Darmstadt, 2019
D17

Oliver Schwahn: *On the Efficient Design and Testing of Dependable Systems Software*
Darmstadt, Technische Universität Darmstadt
Tag der mündlichen Prüfung: 29.03.2019

Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
URN: urn:nbn:de:tuda-tuprints-85772

On the Efficient Design and Testing of Dependable Systems Software

by

Oliver Schwahn

Erklärung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 15. Februar 2019

Oliver Schwahn

Abstract

Modern computing systems that enable increasingly smart and complex applications permeate our daily lives. We strive for a fully connected and automated world to simplify our lives and increase comfort by offloading tasks to smart devices and systems. We have become dependent on the complex and ever growing ecosystem of software that drives the innovations of our smart technologies. With this dependence on complex software systems arises the question whether these systems are dependable, i.e., whether we can actually trust them to perform their intended functions. As software is developed by human beings, it must be expected to contain faults, and we need strategies and techniques to minimize both their number and the severity of their impact that scale with the increase in software complexity.

Common approaches to achieve dependable operation include fault acceptance and fault avoidance strategies. The former gracefully handle faults when they occur during operation, e.g., by isolating and restarting faulty components, whereas the latter try to remove faults before system deployment, e.g., by applying correctness testing and software fault injection (SFI) techniques. On this background, this thesis aims at improving the efficiency of fault isolation for operating system kernel components, which are especially critical for dependable operation, as well as at improving the efficiency of dynamic testing activities to cope with the increasing complexity of software.

Using the widely used Linux kernel, we demonstrate that partial fault isolation techniques for kernel software components can be enhanced with dynamic runtime profiles to strike a balance between the expected overheads imposed by the isolation mechanism and the achieved degree of isolation according to user requirements. With the increase in software complexity, comprehensive correctness and robustness assessments using testing and SFI require a substantially increasing number of individual tests whose execution requires a considerable amount of time. We study, considering different levels of the software stack, if modern parallel hardware can be employed to mitigate this increase. In particular, we demonstrate that SFI tests can benefit from parallel execution if such tests are carefully designed and conducted. We furthermore introduce a novel SFI framework to efficiently conduct such experiments. Moreover, we investigate if existing test suites for correctness testing can already benefit from parallel execution and provide an approach that offers a migration path for test suites that have not originally been designed for parallel execution.

Zusammenfassung

Moderne Computersysteme, die immer intelligentere und komplexere Anwendungen ermöglichen, durchdringen unseren Alltag. Wir streben eine vollständig vernetzte und automatisierte Welt an, um unser Leben zu vereinfachen und unseren Komfort zu erhöhen, indem Aufgaben auf intelligente Geräte und Systeme verlagert werden. Wir sind von dem komplexen und ständig wachsenden Software-Ökosystem abhängig, das die Innovationen unserer intelligenten Technologien vorantreibt. Mit dieser Abhängigkeit von komplexen Softwaresystemen stellt sich die Frage, ob diese Systeme zuverlässig sind, d.h. ob wir tatsächlich darauf vertrauen können, dass sie ihre beabsichtigten Funktionen ausführen. Da Software von Menschen entwickelt wird, muss davon ausgegangen werden, dass sie Fehler enthält, und wir benötigen Strategien und Techniken, um deren Anzahl und Schweregrad zu verringern, die mit der zunehmenden Komplexität skalieren.

Übliche Ansätze, um einen zuverlässigen Betrieb zu erreichen, umfassen Fehlerakzeptanz- und Fehlervermeidungsstrategien. Die Ersteren tolerieren Fehler, wenn sie während des Betriebs auftreten, z.B. durch Isolieren und Neustarten fehlerhafter Komponenten, während die Letzteren versuchen, Fehler vor dem Einsatz des Systems zu entfernen, z.B. durch Anwenden von Korrektheitstest- und Softwarefehlerinjektionstechniken (SFI-Techniken). Vor diesem Hintergrund zielt diese Dissertation darauf ab, die Effizienz der Fehlerisolierung für Betriebssystemkernelkomponenten zu verbessern, die für einen zuverlässigen Betrieb besonders wichtig sind, und die Effizienz dynamischer Testaktivitäten zu verbessern, um der zunehmenden Komplexität von Software Rechnung zu tragen.

Wir zeigen, dass Techniken zur partiellen Fehlerisolierung für Kernel-Softwarekomponenten durch dynamische Laufzeitprofile erweitert werden können, um den erwarteten Overhead durch den Isolationsmechanismus und den erreichten Isolierungsgrad gemäß den Benutzeranforderungen zu balancieren. Mit zunehmender Software-Komplexität erfordern umfassende Korrektheits- und Robustheitsbewertungen mit Korrektheitstests oder Software-Testverfahren und SFI eine wesentlich höhere Anzahl von Einzeltests, deren Durchführung einen erheblichen Zeitaufwand erfordert. Wir untersuchen unter Berücksichtigung verschiedener Ebenen des Software-Stacks, ob moderne parallele Hardware eingesetzt werden kann, um diesen Anstieg abzumildern. Wir zeigen insbesondere, dass SFI-Tests von einer parallelen Ausführung profitieren können, wenn diese Tests sorgfältig entworfen werden. Wir führen außerdem ein neues SFI-Framework ein, um solche Experi-

mente effizient durchzuführen. Darüber hinaus untersuchen wir, ob vorhandene Testsuites für Korrektheitstests bereits von der parallelen Ausführung profitieren können und bieten einen Ansatz, der einen Migrationspfad für Testsuites bietet, die ursprünglich nicht für die parallele Ausführung konzipiert wurden.

Acknowledgments

Working towards a PhD is a long and bumpy journey. I could never have made it this far without the support and help of the many great people that shared parts of the journey with me and that I met along the way. I am deeply grateful for the great times and the support from all the people at DEEDS.

First of all, I would like to thank Neeraj Suri, my advisor and mentor. Thanks for always supporting me, providing invaluable advice, and sharing your wisdom not only in a professional context but also on matters with which we all struggle from time to time in our daily lives. Although I have not found the pineapple tree yet, I always appreciated your insightful advice. Thank you for always being open to new ideas and for building this great research group where we can all pursue our research interests with great freedom and work together with fantastic people.

I am also very grateful to Karthik Pattabiraman for accepting to be my external reviewer as well as to Stefan Katzenbeisser, Guido Salvaneschi, and Thomas Schneider for being on my committee.

A big thank you to Stefan, my long-term office co-inhabitant and friend without whom I probably would never have started this journey. Thanks for always supporting and believing in me and the work we did together. You always encouraged me not to give up and to continue improving my work. I always appreciated your good ideas and our discussions for shaping our joint projects and papers. Thank you for fixing my, sometimes too lengthy, texts, for spending nights and weekends on papers with me, and for introducing me to the large variety of hop beverages.

Thank you Nico, my young friend and latest office co-inhabitant. Thanks for always being supportive and for the occasional cheer up when I need it. Thank you for spending nights and weekends with me working on papers and broken implementations. Hacking on code and finding creative solutions to arcane technical problems together with you was always fun. Thanks for the great discussions, for lifting the spirits after long days of work, and our after work and weekend sessions.

Thank you Habi, my friend and office co-inhabitant since ancient times, for taking breaks when playing the drums. Thanks for our insightful discussions on work, life, love, and technology. Understanding The Curve and The Peak brought me closer to enlightenment. Brainstorming with you was always productive fun, with you adding the formal and me adding a more practical perspective. Thanks for always being supportive and tolerating my many quirks.

Thank you Tsveti for letting me take over your seat in the office; it greatly influenced my life for the last five years. Thanks for always being helpful and full of energy. I enjoyed our technical discussions and the joint work with students. Of course, our conversations over a cup of coffee were also always a delight.

A big thanks to Sabine for always being supportive and helping out whenever possible, especially with the paperwork! I always enjoyed our talks over a cup of coffee in the morning. Thanks for bringing Haley to work, she made me smile and provided a good morning workout. Thank you Ute for always helping me out. I enjoyed our crazy talk on random topics of daily life. We had great times tackling the technical difficulties and server breakdowns together. Let's hope we never have to rely on that backup!

Thank you Salman for our nice talks and supporting me with my karma. It's great that you finally joined in for our black and delicious coffee after lunch. Thanks Heng for fun times. It's always cheering up talking to you and hearing about your unique point of view. Thank you Patrick for being a nice and supportive guy, although you resist drinking coffee with me. Thanks Yiqun for always being right on time for a Mensa-tional lunch and reminding all of us not to miss it.

I would also like to thank all my other co-authors for the great work on joint projects and papers and all the productive discussions we had over the years. Thank you Roberto, Domenico, and Suman. Thanks to the great students I had the pleasure to work with! Thank you Fabsi for helping out with my overly complicated experiment setups. Another thanks to Alex for constructing overly complicated, but POSIX compliant, shell scripts that often worked as intended. Thank you Paddy for diving into the depths of Linux file systems. Thanks Arun for tackling outdated LLVM versions and nasty libraries.

A special thanks to some former DEEDSians. Thank you Thorsten for the great and productive discussions on assorted issues of work and daily life and for the steady st(r)eam of special, and sometimes whimsical, deals and offers from all around the Internet. An extra big thank you for the great evenings that lifted our spirits after long and exhausting days of work. Thanks Hatem for always being understanding and humorous, and for tolerating my (typical?) German quirks. I enjoyed our conversations and our after work activities. Having after lunch coffee with you was also fun; especially, since I finally managed to convince you of the pleasures of drinking your coffee black. Thanks Daniel for introducing me to the Grand Giraffe and the diverse applications of bananas for comparing scales.

There have been more former DEEDS members that I will never forget. Thanks to Ahmed, who was an office co-inhabitant of mine for a short period, for enlightening discussions about Eigenvalues. I was very impressed that your little cactus is still doing fine in the other office. I also want to thank Kubi for wild and sometimes exhausting discussions on cultural differences and similarities, politics, engineering, and many many more topics. We always had good times riding the train home

together in the evening and, of course, waiting for said train in case of DB humor. Thank you Ruben for nice conversations and lunch time fun. Thanks Jesus, it's always a pleasure talking to you, even if it's not about Doctor Who! Another thanks to Zhazira and Giancarlo. Our time together was short, but nonetheless appreciated.

I would also like to thank my friend Christian, we had quite an intense ride together! It was great for as long as it lasted. Always keep the Rock'n'Roll in your heart, and remember that in the end everything will be fine. Thanks to all the other great people and friends I have been bumping into and connecting with throughout my journey. The people you meet along the way make the journey a great experience, especially in times of trouble and doubts. Thank you Jannik, Christian (the other one), Markus, Magic Michi, Thommy, and all the others.

Finally, I would like to thank my family from the bottom of my heart! I could never have made it without your love and support. You were always there for me when I needed you, no questions asked. I am forever grateful! A big thank you to my Mom and Dad for always being understanding and supportive in every possible way. Thanks Brigitte and Fritz, I have always enjoyed paying you a spontaneous visit. The Kischl were always delicious (the holes give the additional character) and the funny sayings often cheered me up. Thank you Mef and Marc, BBQs, Erdbeerbowle, and our discussions have always been fun. Who would have thought that chicken on a can of beer is so delicious! Thanks Mätt and Stefanie, may the honey production never stall. I like to remember all our adventures, for instance, in Ye Old Carriage Inn. Thank you Christine and Manfred, I have always enjoyed our discussions and having good wine (or rum) with you guys. Set sail and full speed ahead!

Oliver Schwahn
Darmstadt, March, 2019

Contents

1 Introduction

Computing systems and the services they provide have become ubiquitous in our daily lives. They take on various shapes and sizes, from small embedded systems to large scale servers, perform a multitude of tasks, and are continuously updated with new functions, often by means of software updates. We strive for a fully connected and automated world in which systems and devices function and interact autonomously for simplifying our lives and increasing comfort. This vision is driven by an ever growing ecosystem of software that enables the increasingly complex functions and applications we demand. The Internet of Things (IoT) is one of the latest manifestations of this trend where all kinds of devices and physical objects, which were traditionally not interconnected, are infused with technology and software to enable them to interact with their environment, other devices, and online services. We rely on personal smart devices, such as smartphones and smartwatches, being interconnected communication hubs with permanent connections to the Internet, not only to access the functions of the IoT, but also to drive and organize our daily lives. The worldwide number of connected devices is growing rapidly [Cis18], with the estimated number increasing from about 17 billion devices in 2017 to over 27 billion devices in 2022 with over 50 % of connections being directly between devices (machine-to-machine). Moreover, we increasingly make use of smart, AI-powered (artificial intelligence) voice assistants, for instance, to control functions in smart homes, whose market is continuously growing with the top five areas in 2018 being security and safety systems (e.g., door locks), audiovisual (e.g., connected speakers), smart energy, and software platforms [AY18]. All these smart technologies involve an extensive amount of software that is orchestrated in a stack of software components with the upper layers depending on the lower ones.

This trend of computerization and automation by means of software continues in the area of safety critical systems including applications in medical, traffic control, railways, aviation, spaceflight, and automotive. For instance, in 2009, certain commercial airplanes required 6.5 million lines of software code to operate and premium-class automobiles were estimated to require around 100 million lines of code executing distributed among 70 to 100 processing units [Cha09]. In 2019, the amount and complexity of software in cars alone likely increased manyfold with cars offering many software-implemented features such as drive/steer by wire and advanced driver assistance like traffic-aware cruise control, automatic lane keeping,

and automatic emergency braking. As the automotive industry is on the verge of developing self-driving cars, complexity in this area will increase even more.

In order to operate correctly and satisfy user expectations, computing systems have to provide a certain level of performance and responsiveness. For many systems this means to provide responses to user requests within a certain amount of time to meet user expectations [Nie94]. But in the case of real-time systems, responses heave to be provided or certain actions be taken within well specified time frames [Wan17]. As innovative applications require more and more computing power and the performance of individual processing units (CPUs) is already at its peak, mainly due to physical constraints, hardware has become highly parallel [Rup18] and provides multiple processing units, i.e., multi-core CPUs. Consequently, modern systems and their software are being designed and adapted to make use of the available parallel hardware, thereby further increasing their complexity.

In most cases, an Operating System (OS), being in the lowest levels of the software stack, manages the hardware and provides software services that simplify the development of software at higher levels of the stack that implements the desired functionality of our devices. Linux is a prominent example of a versatile OS kernel that is used in many different application scenarios spanning embedded systems, desktops, servers, and supercomputers. In addition to being the underlying kernel for the Android smartphone OS, which, at the end of 2018, had over 86 % market share [IDC19], Linux is running over 35 % of the top 10 million websites [QSu19b] in early 2019. To support these versatile usage contexts the Linux code base grew from 9.7 MSLOC[1] in July 2011 (Linux 3.0) to 17.4 MSLOC in December 2018 (Linux 4.20), which corresponds to an increase in code base size of 1.8× in 7 years. Remarkably, 56 % (5.5 MSLOC) of code in 2011 was device driver code, which enables the OS to utilize different hardware devices (e.g. hard drives, network adapters, and peripherals), whereas in 2018, 66.1 % (11.5 MSLOC) of the code base was dedicated to device drivers, which means that the amount of code required to support the growing variety of hardware grows even faster (2.1×).

With our increasing dependence on complex software and its correct composition and orchestration, the question arises if we can actually and justifiably trust these complex systems to operate correctly and perform the expected tasks, i.e., are they dependable? As the software is developed by human beings and the development process itself is subject to many constraints such as development cost budgets in commercial contexts, software must be expected to contain defects (often termed "bugs"). Moreover, software re-use has become common, e.g., the

[1] MSLOC means *million source lines of code* and measures the number of physical non-empty, non-commented source code lines. The numbers presented have been generated using David A. Wheeler's SLOCCount tool. The Linux kernel sources that were counted have been retrieved from the official *linux-stable* Git repository at `git.kernel.org` using the Git tags *v3.0* and *v4.20*.

usage of (commercial) off-the-shelf ((C)OTS) components, which most likely contain unknown defects, and the integration of re-used software in different application contexts can have unanticipated side effects. Software defects have a wide range of consequences. They can lead to simple annoyances when a user has to reboot their smartphone, but they can also lead to severe financial losses (e.g., when spacecraft are lost [Lev04]) and, in case of safety critical systems, even cost human lives [LT93]. For safety critical systems, international standards, such as IEC 61508 [Int10] and ISO 26262 [Int11], are in place that prescribe development processes and quality assurance measures to limit safety risks.

To minimize both the number and impact of defects in deployed software, i.e., to increase its dependability, various approaches are usually combined. One general perspective is to limit the impact of faults or defects by compartmentalization and isolation such that the effects of such defects are contained within one compartment. Another perspective is to improve the software quality before deployment such that the number and severity of defects is reduced and the robustness of the software is increased.

In many systems, certain parts or components within the software stack are more critical than others, and the latter should not be able to hinder the former in performing their intended function, i.e., critical components should be isolated from uncritical ones. In an OS, an application executing in low-privilege user mode must not block an OS service executing in high-privilege kernel mode. However, the failure of a critical software component such as the OS kernel leads to a failure of the system as a whole. Hence, it is desirable to keep the amount of software that executes in such a critical context to a minimum [Rus81], thereby evening out the increase of complexity for these critical software components. Unfortunately, complex systems have often been designed in a monolithic way without isolation between critical and non-critical components or with critical components being larger than necessary. The Linux kernel is a good example for this design as it executes its over 17 MSLOC in kernel mode although certain parts such as device drivers have been shown to contain considerably more defects than other kernel code [Cho+01; Pal+11], which makes them attractive candidates for isolation in user mode as is done in microkernel OS designs [Kle+09; Lie]. However, retrofitting an originally monolithic design with additional isolation capabilities introduces additional runtime overhead, potentially decreasing system performance to an unacceptable level. There is usually a trade-off between achievable degree of isolation and performance that requires careful balancing.

As performance considerations usually impose limitations on the achievable degree of isolation, complementary techniques are still needed to find and remove defects. Software (correctness) testing is a time consuming part of the software development process and can be considered a quality assurance activity. Its goal is to find, and ultimately remove, defects in the software under test (SUT) [Bei03;

MSB12]. In dynamic correctness testing the SUT is executed and exposed to known inputs and the resulting responses (outputs) are compared to the expected responses. This process itself is usually automated using software (test harness). Each pair of inputs and responses is considered a test case. Complex software requires a large amount of test cases for thorough testing, with many software projects including more code dedicated to testing than for the actual application logic [GVS17]. Test execution by itself becomes a bottleneck with increasing numbers of test cases as running more tests naturally requires more time, slowing down the already time consuming testing process even further.

Fault Injection (FI) [HTI97] is a complementary technique for assessing and improving the dependability of a system under faulty conditions. FI artificially introduces faults, e.g., bit flips in main memory, while observing the reaction of the system. FI is well established for assessing fault tolerance mechanisms and is prescribed by international standards for safety-critical systems such as ISO 26262 [Int11] for automotive systems. Whereas traditional FI aims at emulating hardware faults, software fault injection (SFI) [DM06] aims at emulating faults or defects within the software itself. SFI emulating representative residual faults [CN13; Nat+13], i.e., defects that have not been found during testing and are therefore present in deployed software, is especially useful for a realistic assessment when software is re-used and OTS components are integrated. However, similarly to classic testing, the increased complexity of software necessitates an increasing number of FI experiments for a comprehensive assessment, slowing down the overall software development pace.

On this general background, this thesis

1. develops a profiling driven approach for the bi-partitioning of in-kernel software components to explore the trade-off between runtime performance and degree of code isolation,

2. develops approaches and techniques to reduce the execution latencies for SFI tests by avoiding unnecessary overheads caused by isolation mechanisms, avoiding redundant work, and exploiting parallel hardware, and

3. investigates the potential for parallel testing of software for the reduction of test execution latencies within a popular Linux-based OS ecosystem and proposes strategies to further benefit from parallel hardware.

The developed approaches and techniques aim at improving the efficiency of dependability improving activities during software development and of retrofitted isolation for monolithic designs. The studies and experiments to develop and evaluate these techniques have been conducted on software that can be broadly considered to be at the lower levels of the software stack, such as the Linux kernel, as these components are essential to the dependability of a system as a whole.

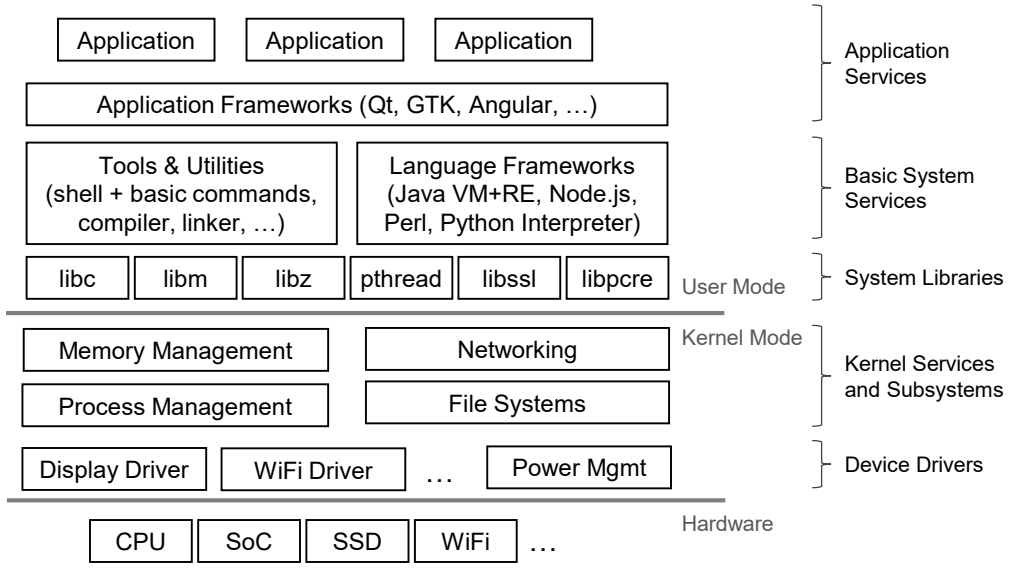


Figure 1.1: Illustration of a Software Stack Including Hardware Layer

In the remainder of this chapter, we give some background on the complexity of the software stack in Section 1.1 and discuss some more background on dependable systems and software in Section 1.2.

1.1 The Software Stack

In this section, we detail the notion of software stack and software components that underlies the work presented in this thesis. Moreover, we argue why we consider the lower levels of the stack especially important and interesting for research.

Figure 1.1 is a simplified illustration of an example software stack. For instance, the stack for a web service executing on a typical multi-core x86 server may look like this. At the very bottom, although not pure software in the strict sense[2], is the hardware platform on top of which the software runs. Immediately above the hardware layer, the operating system (OS), more precisely the OS kernel, is located. In general, the OS provides an abstraction from the hardware as well as management and coordination of resources. It simplifies application development and enforces security and resource usage policies. Basic system libraries (collections of software functions) and utilities are usually also considered as being part of the OS. At the very top of the stack are the applications, i.e., the functionality that the system is intended to provide to the end users, for instance, the content

[2]Most hardware devices contain software of their own, so called firmware, to control the lowest level of hardware functionality.

management system of a website or an office application with graphical user interface. Applications execute within one or more processes, possibly using concurrent threads. The software layers between the OS and the applications are often referred to as middleware. Middleware provides a multitude of functions and frameworks that ease the development of application software beyond the basic services of the OS itself. Examples for middleware include language runtimes for interpreted languages such as Java, Python, Perl, or JavaScript, but also complete frameworks for application development such as Qt[3], GTK[4], or Angular[5].

Privilege Levels

The illustrated hardware platform supports two hardware-implemented privilege levels for software execution that the OS makes use of: user mode and kernel model. Software executing in kernel mode has the highest privileges and, therefore, has unrestricted access to all resources of the system, including main memory and hardware devices. The OS kernel executes in this mode which makes it a highly critical component as a malfunction within the kernel can easily bring down the system as a whole. All other software, i.e., everything except the kernel, executes in user mode with restricted privileges with the consequence that user mode software must rely on OS services to perform certain actions, e.g., access files or hardware devices, which allows the OS to enforce security and resource usage policies. Typically, user mode software invokes kernel services by performing system calls that transfer control to the kernel, which then acts on behalf of the calling software. System calls cross the boundary between user and kernel mode and imply a performance penalty as additional actions must be taken by both the hardware and the kernel. The processes that implement the applications are usually isolated from each other with separate memory address spaces, which are enforced by the OS with the help of hardware (memory management unit or MMU). Note that the described architecture with two (or sometimes even more) hardware privilege levels is highly relevant as many platforms (e.g., x86, ARM, RISC-V) make use of variations of it. However, other architectures, for instance, without separate hardware privilege levels and/or memory address space separation, are often found in embedded devices using simple micro controllers.

Monolithic and Micro OS Designs

The illustrated software stack assumes a monolithic OS architecture, i.e., all services of the OS execute together as part of the kernel at the highest privilege level and in the same memory address space. The Linux kernel is a prominent example

[3] https://www.qt.io
[4] https://www.gtk.org
[5] https://angular.io

for this architecture. An advantage of this design is that kernel components can invoke each other's services by means of simple function calls and large amounts of data can be exchanged very efficiently without the need of copies by passing memory addresses. Additional overheads of crossing the privilege boundary are avoided. As already hinted at above, this design has the disadvantage of a large and complex code base executing in privileged mode. Any software defect located in this code base can potentially harm the system, for instance, by overwriting memory areas of other kernel components or user applications and even damage hardware by sending invalid commands. To support the ever growing diversity of hardware, OSs in general rely on special extension components termed device drivers (also loadable kernel modules in Linux) to establish the interaction between the core kernel and the specific devices. Such device drivers can often be loaded and unloaded on demand once new devices are connected or disconnected from the system. For Linux, it has been shown by means of static code analysis that device driver code overall contains more defects than other parts of the kernel code [Cho+01; Pal+11], which is not surprising given the sheer amount (66 % of the code base at the end of 2018) of driver code and the variety of supported devices..

An alternative OS design is based on the idea of microkernels [Kle+09; Lie]. Such designs follow the philosophy to minimize the amount of code inside the kernel, and thereby running in privileged mode, to a bare minimum. Typically, all device drivers and most other OS services, e.g., networking and file systems, execute in user mode inside ordinary processes with separate memory address spaces. By minimizing the amount of code executed in kernel mode, the likelihood of that code including software defects is decreased accordingly and the so called trusted computing base (TCB) [Rus81], i.e., the code one has to trust to work as intended, is reduced. Microkernel designs are an extreme departure from classic monolithic designs that have grown and been in use for years and require the rewrite of large portions of software in the lower layers of the software stack. For instance, device drivers, making up a majority of code in OSs, have to be rewritten, as they are moved higher in the software stack. The same is true for many system libraries that closely interact with the OS kernel. In addition, microkernel-based systems have historically often suffered from degraded performance for certain workloads compared to monolithic designs, which hindered their adoption outside of specialized domains such as embedded systems.

The middle ground between the extremes of monolithic and microkernel designs is to retrofit the capability to execute certain kernel components or parts of them in user mode rather than inside the kernel [Gan+08; RS09]. Such an approach has the advantage of backwards compatibility, i.e., most or all of the existing code can be re-used. However, in order not run into prohibitive performance bottlenecks with such a design, the trade-off between the amount of code that is removed from kernel mode and the achievable performance, while remaining backwards

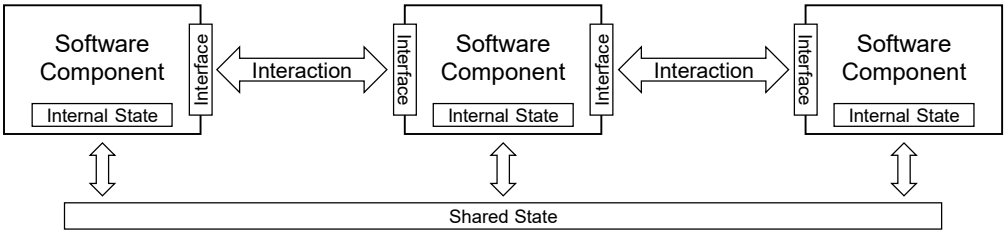


Figure 1.2: Illustration of Interacting Software Components

compatible, must be carefully assessed and balanced. The imposed overhead of such a solution depends on the usage profile of the targeted system, i.e., how heavily the isolated component is actually used in operation.

Software Components

The software stack is built on top of the hardware by composing and orchestrating different software components to achieve the overall desired functionality of the system. A software component is a module or collection of code that bundles a set of related functions. Often, such components can be internally subdivided into finer-grained components. For instance, from a high level point of view the OS can be seen merely as a software component in the overall system, but at closer inspection, the OS by itself can be seen as a collection of interacting components, such as device drivers.

Figure 1.2 provides a component-based view on a three component system. The components interact through interfaces with each other, often have internal state (e.g., local variables), and possibly also have shared state (e.g., global variables and common files). Interfaces between components can take on many forms in practice, depending on the nature of the software and the usage scenario. For instance, the interface to a code library is often specified by the set of functions that are declared public and are hence invokable from other components. However, an interface can also include less explicit mechanisms, for instance, one component may read and write a global memory variable provided by another component. In general, explicit and well documented interfaces are preferable as this approach enables portability, code re-use, and the integration of OTS components, which is often economically attractive. In such a scenario, one component can be replaced with another one that implements the same interfaces but, for instance, performs better in the intended usage scenario. A prominent example is developing code against the specification of an interface rather than for the use with a specific component. For instance, applications that have been developed against the POSIX [IEE18] specification rather than a specific OS implementation can often be easily ported

across hardware and OS platforms, as long the targeted platform provides a POSIX implementation.

Ideally, components are as independent from each other as possible (low coupling), do not share state, and one component cannot adversely influence the correct function of another component, e.g., by corrupting its internal state. This is, for instance, asymmetrically the case between an OS kernel and the applications running on top of it as the kernel controls the applications. Hence, the kernel can terminate applications, but applications cannot uncontrollably shut down the kernel, at least in a perfect world without malfunctions, faults, and defects. An interesting aspect that we will discuss later in Section 1.2 is how defects in one component affect other components and, consequently, the overall system and its function.

Performance-centric Software

Performance and efficiency is a strong driver behind system design and has driven many areas of technology for years. For instance, CPU designs have always been adapted to enable ever increasing single core speed and when increasing single core speed became infeasible, more and more additional, both physical and logical, cores were (and still are) added [Rup18] to further increase performance. Contemporary smartphones can easily incorporate 8 [Qua18] and desktop PCs even 12 [Adv19] and more physical CPU cores. In order to harness all this computing power, software across the whole stack has to evolve as well to make use of concurrency and parallelism of modern platforms, increasing complexity and potentially raising new problems [Coro8].

The Software at lower levels in the stack is particularly critical to good performance as it provides the basic services and functions for the application software. Interestingly, many of the lower software layers involve software that is developed using the C programming language, likely because it has been used in practice for a long time and is therefore very mature, but also because it allows for an efficient and predictable use of available hardware resources as it provides programming abstractions that are not too far away from how the underlying hardware is operating. OS kernels are often developed in C, with the Linux kernel being a prominent example. However, C is also popular in other fields. A study of 100 000 Github projects [Bis+13] showed that C was the most widely used language as over 60 % of the code in the studied projects was written in C and it was also the most popular language among developers (22 %). C is also prominently used in embedded systems contexts. A survey of embedded systems developers in 2018 showed that about 70 % [Bar18] of participating developers used C as their primary language. Moreover, system that have to process many requests in a short amount of time and where efficiency is therefore paramount also often rely on the C language. For

instance, about 85 % of the top 10 million website run on server software (Apache and Nginx) that is written in C [QSu19a] at the beginning of 2019. We will also show later in Chapter 4 that C is in wide use within the Debian OS ecosystem.

The available hardware resources should also be leveraged during the software development process. Parallel hardware requires thinking about the parallelization of the automated portions of the software development process as well. If we can develop software faster, software updates can bring new and smarter features to our daily lives faster.

1.2 Dependable Software

In this section we discuss the notion of dependability and give background information on related concepts and techniques that are of interest for this thesis. This discussion is largely based on the taxonomy proposed by Avizienis et al. [Avi+04].

With the growth of software stacks in size and complexity, involving re-used components from different sources, comes the question whether we can depend on the systems we build. Dependable software and systems are those that have the ability to provide services that can be justifiably trusted [Avi+04]. The key to this definition is that it is necessary to justify this trust in a system. An alternative point of view is that a system can be considered dependable if it has the ability to avoid service failures that occur more frequently or have more severe consequences than is deemed acceptable [Avi+04]. With this notion of acceptable failures comes a criterion to decide whether a system is dependable or not as one can assess the system according to a stated definition of acceptable, which is part of the dependability specification of a system.

Dependability can be considered a higher level concepts that is composed of multiple system attributes:

- **Availability**: The system is ready to provide correct service. Availability is usually expressed in terms of the proportion of time a system is in operational state and can accept service requests. For instance, if a given system is supposed to operate in a period of 12 hours, but it is only operational for 6 hours during that period, its availability is 50 %.

- **Reliability**: The System provides correct service continually. Reliability contains the notion of continuity, i.e., the provided service must be available for a sustained time period. For instance, if a system fails often but only for short periods of time within the time frame it is supposed to be operational, it has high availability but low reliability. Reliability is often specified as mean time to failure, i.e., the average time between consecutive failures.

Figure 1.3: The Threats to Dependability and Their Relationship

- **Safety**: The system does not harm its users or environment. A safe system is designed to prevent severe consequences both during its normal operation as well as in case of failures. Such systems enter a safe state if erroneous conditions are detected. Safety-critical systems such as in automotive applications have a particular focus on this attribute.

- **Integrity**: The system cannot be improperly altered. Neither by accident nor on purpose can the system be changed to add, remove or alter implemented services without being detected. For instance, a piece of hardware may be sealed in especially durable enclosures or software may contain checksums or cryptographic signatures to detect code alterations before execution.

- **Maintainability**: The system can be modified and repaired as necessary. In case of malfunctions the system is accessible to repair activities, for instance, individual components can be replaced with spare parts (in the case of hardware) or updated versions (for software). New features can be added or existing ones modified with ease, e.g., by means of software updates.

Dependability includes the notion of delivering *correct* service. Correctness means that the system indeed implements the functions that it is *intended* for, which are stated in the system's functional specification. A system is robust if it is able to gracefully handle inputs and environmental conditions that are beyond its functional specification.

The Threats to Dependability

If a systems stops delivering correct service, we speak of a service failure. Such failures are characterized by the exhibited failure modes, which can be ranked according to their respective severity and be classified according to their domain, detectability, consistency, and consequences. For instance, failures can be related to service content and timing (domain), can be signaled or unsignaled, can be consistent or inconsistent (Byzantine), and can have wide ranging consequences from minor to catastrophic. Beyond failures that result from a service not adhering to its specification, failures also occur if the service deviates from its *intended* function. This is the case if the specification itself contains mistakes or is incomplete with respect to what was intended. As intention is difficult to precisely capture and express, this is where robustness issues arise.

In general, failures are caused by a chain of events as illustrated in Figure 1.3 on the previous page. The underlying cause of a failure is a fault, which is a flaw or defect within or external to the system, e.g., in its design or program code (bug). A fault remains dormant until it is activated (triggered), e.g., a defective piece of code is executed. The activation of the fault leads to an error in the system state, e.g., a wrong value in some program variable. Note that in order for an external fault to cause an error within the system, the presence of an enabling internal fault is a necessary precondition. If the error propagates to the interface of the system and becomes observable to external entities (users), i.e., leads to a deviation from correct service, a failure occurs. Such propagation may occur, for instance, when a program uses a corrupted variable to perform further computations whose results are part of the delivered service. Once an error has occurred, it may be detected or undetected, with the latter being a latent error. The presence of an error does not necessary lead to a failure as propagation to the interface is not guaranteed. An error may reside in parts of the system state that are not related to the direct delivery of correct service or an error may be overwritten before it can propagate. The interplay of faults, errors, and failures becomes more complex when multiple interacting systems or components are considered where a system A depends on the services of another system B to deliver its own service. In such a scenario, the chain of events may extend across multiple systems. A fault in system B may get activated, leading to an error, which propagates and leads to a service failure. This failure becomes an external fault for system A. Due to an internal fault of system A, e.g., absence of input value validation, this may lead to an error in system A and ultimately result in the failure of A.

Faults generally fall into three different (overlapping) groups: development faults, which include all faults being introduced in the development phase, physical faults, which include all faults that affect hardware, and interaction faults, which include all external faults. Furthermore, they can be categorized, among others, according to when they are introduced, during development or once the system is in operation, whether they are internal or external of the system, whether they occur in hard- or software, and whether they are permanent or transient. Note that all development faults are permanent faults. The typical notion of software bug or defect falls into the category of permanent, internal, software faults introduced during development. As human beings are an integral part of the development and maintenance process, all systems and components, including their specifications, must be assumed to contain faults to some extent, which is why we need systematic approaches to deal with them and mitigate their effects.

The Means for Dependability

In order to build dependable systems, i.e., systems in whose services we can justifiably trust, many approaches and techniques have been developed over the past decades. These techniques commonly fall into one of the following four distinct categories, commonly known as the means for dependability [Avi+04]. They focus on how to deal with faults, which are the underlying causes for failures, to achieve dependable operation.

- **Fault Prevention**: Prevent the introduction of the fault in the first place. By employing good engineering practices and adhering to development standards, rules, and processes, the introduction of faults is minimized during the development process. For instance, a suitable programming language for the problem at hand is chosen, state-of-the-art development tools are used, and developer qualification is improved.

- **Fault Tolerance**: Avert service failures despite the presence of faults. During system operation the occurrence of faults is expected and error detection and recovery mechanisms are employed. Often such schemes include redundancy of components for both the detection and recovery from errors.

- **Fault Removal**: Reduce the amount and severity of faults. Faults are identified during the development process by means of static, e.g., model checking or theorem proving, and dynamic, e.g., symbolic execution or testing, approaches. Identified faults are then removed before the system is deployed.

- **Fault Forecasting**: Estimate the number of faults, their future incidence and consequences. Using statistical modelling and data on historic system behavior as well as testing techniques, faults and their severity are estimated.

These four means can be grouped into two fundamental approaches or points of view: fault avoidance and fault acceptance. Both fault prevention and removal attempt to avoid faults, i.e., construct systems that are free from faults when in operation. Fault avoidance approaches aim at the development process and, in the case of software, the improvement of the design and code quality. In contrast, fault tolerance and forecasting acknowledge the fact that there will be faults when a system is in operation and plan for their occurrence. Such approaches try to estimate and limit their impact by statistical modelling and adding additional mechanisms that prevent or mitigate service failures, often relying on component redundancy.

The scenarios and techniques considered in this thesis refer to both aspects as we consider a fault containment (isolation) scenario for kernel software components (cf. Chapter 2), which falls into the category of fault tolerance and therefore acceptance,

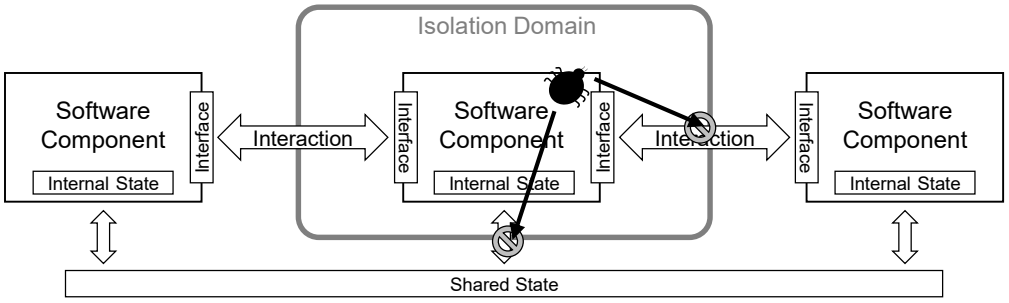


Figure 1.4: Illustration of Interacting Software Components with Isolation

as well as dynamic testing scenarios, which fall into the category of fault removal and therefore avoidance, namely fault injection for robustness assessments (cf. Chapter 3) and correctness testing (cf. Chapter 4).

Fault containment or isolation techniques aim at preventing error propagation beyond the boundaries of an isolation domain in case a dormant fault is activated. Figure 1.4 illustrates an example of a system with three interacting software components where a presumably faulty component is locked into its own isolation domain. To contain faults and their effects within the isolation domain, the isolation mechanism has to interpose on all interactions between domains as well as on accesses to shared state to prevent state corruption. Indeed, isolating components that share state from each other proves difficult and imposes noticeable overheads. Once an error detection mechanism detects an error within the isolation domain, the isolated component must be recovered to restore correct operation, e.g., by restarting it.

By applying correctness testing to the faulty component, the contained (software) fault could be found by rigorous testing if the test suite contains a test case that is able to reveal the fault. If the fault leads to a deviation from *specified* (in contrast to *intended*) correct service, a comprehensive test suite may contain a fault triggering test input. The fault revealing test case can then be leveraged for debugging purposes and the failure causing fault can ultimately be removed. If no isolation technique is employed and the fault cannot be identified by correctness testing, possibly the fault is not covered by the specification or the test suite is not comprehensive. Software fault injection techniques can then be applied to assess the robustness of the other two components against faulty behavior of the faulty component. In this case, known faults would be injected in the middle component and the interaction between the three components be observed to assess if the error caused by the activation of the fault in the middle component propagates to the other components possibly leading to their failure. In case such error propagation is observed, a potential robustness issue has been identified that can be further analyzed and ultimately be repaired to improve the robustness

of the affected components. We have investigated error propagation in different application contexts in earlier work, namely in mixed-criticality automotive systems [Pip+15] and within OS kernels [Cop+17]. Both the comprehensive correctness testing as well as the comprehensive robustness assessment using fault injection requires a large number of individual tests, which is a time consuming process, especially if both techniques are combined.

In contrast to our example, it is generally unknown if and where real systems contain faults. Therefore, one cannot exactly quantify which benefits applying either of these techniques would have before actually applying them. As we are interested in building dependable systems, and all the above mentioned techniques can be used together as building blocks, they should be used in conjunction. Therefore investigating strategies to improve their efficiency is important in order to overcome slow execution times that might otherwise prohibit their usage.

1.3 Research Questions and Contributions

This thesis is driven by the research questions stated below and the investigation of said questions led to the contributions that are summarized below as well. The common theme underlying all these questions is the desire to improve the dependability of our complex software systems without harming their usability or slowing down their development process. To that end, the first question investigates fault containment (isolation) for OS level software. The second question aims at improving the efficiency of dependability assessments using SFI for both OS-level as well as higher level software. The third question investigates if similar techniques that we applied for SFI can be used to improve the efficiency of (correctness) testing during software development.

Research Question 1 (RQ 1): Can runtime profiling be leveraged for the partitioning of in-kernel software components to increase code isolation while balancing performance overhead?

Many OSs employ a monolithic design, in which in-kernel software components, such as device drivers, are not isolated from each other. Consequently, the failure of one such component can affect the whole kernel. While microkernel OSs provide such isolation for large parts of the OS, they have not been widely adopted, due to performance and compatibility related issues, and monolithic kernels, such as Linux, are still prevalent. Approaches offering a middle way between the full isolation of microkernels and the absence of isolation in monolithic designs have been proposed. Such approaches partition the targeted component and isolate only one of the resulting parts. However, these approaches neglect the dynamic usage properties of the targeted components that needs to be taken into account

to find component partitionings that are favorable in terms of code isolation and achievable performance.

Contribution 1 (C 1): Runtime profiling based approach to tailor partitioning to performance needs

Although the proposed approaches for relocating in-kernel code to user mode provide the mechanisms for split mode user/kernel operation of monolithic kernel code, they do not provide guidance on *what* code to execute in which mode. To this end, we develop a partitioning approach that combines static and dynamic analysis techniques to assess the impact of code partitioning decisions on both the degree of isolation and the expected performance overheads in Chapter 2, which is based on material from [Sch+18b]. We make use of dynamically recorded cost data, which we obtain by executing an instrumented variant of the target kernel software component, to model the user/kernel partitioning problem for existing kernel code as 0-1 integer linear programming (ILP) problem and employ a linear solver to obtain partitionings that achieve the desired trade-off between expected performance overhead and the size of the kernel mode code portion for improved isolation. We implemented our approach for the widely used Linux kernel and validate its utility by profiling and partitioning two device drivers and a file system in a case study. We generate a spectrum of partitionings with different balance factors between expected overheads and partition sizes to demonstrate the adaptability of the obtained partitioning to user requirements. Using software fault injection, we also demonstrate the impact of defects depending on whether they are located in the user or the kernel partition and demonstrate the reliability benefits of having larger user partition sizes. This contribution has been documented in the publication "How to Fillet a Penguin: Runtime Data Driven Partitioning of Linux Code" in TDSC 2018.

Research Question 2 (RQ 2): How can parallel hardware be exploited to increase the efficiency of software fault injections?

With the increasing complexity of our software stack, a vast number of SFI experiments are required for comprehensive assessments of the dependability of software components or whole systems. As done in other areas of software engineering, exploiting the increasingly powerful parallel hardware available in virtually all desktop and server machines seems a natural approach to mitigate exploding test numbers and the prolonged execution times they imply. However, parallel execution bears the risk of influencing systems or components targeted for SFI in unexpected ways and thereby subtly changing their behavior, which may lead to a distortion of SFI test results. If SFI test results are not accurate when obtained from accelerated parallel executions, they should not be used to assess the dependability

of systems, especially if said systems are safety-critical, as false conclusions about dependability properties may be drawn.

Contribution 2 (C 2): A framework for increasing the throughput of SFI tests by parallel execution and avoiding redundant work

We investigate whether the parallel execution of SFI tests does accelerate the testing process and whether such parallel SFI tests yield accurate results with respect to traditional sequential test execution in Chapter 3, which is based on material from [Sch+18a] and [Win+15b]. Moreover, we develop techniques to accelerate SFI tests by further means beyond simple parallelization by replication. We first conduct a study of PAIN (PArallel fault INjection) experiments on the Android OS. In this study, we assess the trade-off between achievable increase in experiment throughput and accuracy of obtained results. We indeed identify several causes for significant deviations between sequential and parallel SFI tests and give guidance on how to avoid the invalidation of results obtained from parallel experiments. Our PAIN study makes use of our GRINDER platform for SFI tests that we developed for the PAIN study and that we document in [Win+15a], but do not directly include in this thesis. Applying the insights from our PAIN experiments on the OS level, we develop FastFI, an alternative approach for accelerating SFI tests for FI targets above the OS level. FastFI accelerates SFI testing by avoiding unnecessary overheads caused by isolation mechanisms, avoiding re-executing redundant work, exploiting parallel hardware, and reducing compilation times for faulty versions of the targeted software component. This contribution has been documented in the publications "No PAIN, No Gain? The Utility of PArallel Fault INjections" at ICSE 2015 and "FastFI: Accelerating Software Fault Injections" at PRDC 2018.

Research Question 3 (RQ 3): What is the state of parallel testing for C software and can it be improved to reduce test suite execution latencies?

During software development, testing is a time consuming activity of which the execution of test suites is an important part. With the rise of highly parallel hardware, it is only natural to make use of this computing power to reduce the latency of test suite execution. However, if test suites were not originally designed for being executed in parallel or concurrently, the individual tests may interfere with each other if executed in parallel, which can lead to result deviations compared to sequential execution. To prevent such interferences, each individual test can be provided with an isolated execution environment, but this entails performance overheads that diminish the merit of the parallel execution. As tests evolve together with the software they are meant to test, there is a large amount of testing code, which can be re-used for parallel testing if the individual tests can be orchestrated in a safe and efficient manner.

Contribution 3 (C 3): An assessment of real world C software test suites and an approach for safe concurrent execution of existing tests

We investigate the potential for parallel testing of C software, which is an important building block of most software stacks, for the reduction of test suite executions latencies in Chapter 4, which is based on material from [Sch+19]. We present an analysis of the main software package repository of Debian Buster, which is one of the most widely used Linux-based OS distributions. Our analysis shows that the majority of code contained in the repository is written in C, that no test framework dominates test implementations for C software packages, and that few test suite implementations can benefit from out-of-the-box concurrent execution. Therefore, we develop automated static analyses for existing C test suites to identify test case interdependencies on files and shared global data to identify which parts of a test suite can safely execute in parallel. We design and implement a new test harness to use this information for the safe parallel execution of tests and explore the trade-off between analysis overheads and execution latencies for different parallelization alternatives using processes and threads. We demonstrate the utility of our approach by applying it to nine projects from the Debian Buster software repository. Our results show that test suites in C can benefit from parallel execution, that threads do not perform significantly better than processes, and that our test harness (and likely any specialized test tool) outperforms generic automation tools like `make`. This contribution has been documented in the publication "Assessing the State and Improving the Art of Parallel Testing for C" under submission at ISSTA 2019.

1.4 Publications

The following publications have, in parts verbatim, been included in this thesis.

[Sch+18b] Oliver Schwahn, Stefan Winter, Nicolas Coppik, and Neeraj Suri. "How to Fillet a Penguin: Runtime Data Driven Partitioning of Linux Code". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (Nov. 2018), pp. 945–958. DOI: 10.1109/TDSC.2017.2745574

[Sch+18a] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "FastFI: Accelerating Software Fault Injections". In: *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. PRDC'18. Taipei, Taiwan, Dec. 2018, pp. 193–202. DOI: 10.1109/PRDC.2018.00035

[Win+15b] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. "No PAIN, No Gain?: The Utility of PArallel Fault INjections". In: *Proceedings of the 37th International Conference*

on Software Engineering. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 494–505. DOI: 10.1109/ICSE.2015.67

[Sch+19] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "Assessing the State and Improving the Art of Parallel Testing for C". in: *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019. [under submission]

The following publications are related to different aspects covered in this thesis, but have not been included.

[Win+15a] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. "GRINDER: On Reusability of Fault Injection Tools". In: *Proceedings of the 2015 IEEE/ACM 10th International Workshop on Automation of Software Test*. AST '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 75–79. DOI: 10.1109/AST.2015.22

[Pip+15] Thorsten Piper, Stefan Winter, Oliver Schwahn, Suman Bidarahalli, and Neeraj Suri. "Mitigating Timing Error Propagation in Mixed-Criticality Automotive Systems". In: *Proceedings of the 2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. ISORC '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 102–109. DOI: 10.1109/ISORC.2015.13

[Cop+17] Nicolas Coppik, Oliver Schwahn, Stefan Winter, and Neeraj Suri. "TrEKer: Tracing Error Propagation in Operating System Kernels". In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 377–387. DOI: 10.1109/ASE.2017.8115650

[CSS19] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. "MemFuzz: Using Memory Accesses to Guide Fuzzing". In: *12th IEEE International Conference on Software Testing, Verification and Validation*. ICST 2019. Xi'an, China, Apr. 2019. [accepted]

1.5 Organization

The rest of this thesis is structured as follows. In Chapter 2, we discuss our first research question and develop and evaluate our approach that leverages runtime profiling for the partitioning of in-kernel software components such as device drivers to explore the trade-off between performance overhead and degree of code

isolation. We continue in Chapter 3 with the discussion of our second research question and develop techniques to reduce the execution latencies for software fault injection tests by relying on parallel hardware and avoiding overheads from strong isolation mechanisms and redundant work execution, while maintaining accurate test results. We then discuss our third research question in Chapter 4 and investigate the potential for parallel testing of software to improve test execution latencies within the popular Linux-based Debian OS ecosystem and propose strategies to further benefit from parallel hardware. Finally, Chapter 5 concludes this thesis by providing a summary along with its contributions and key insights.

2 Profiling Driven Partitioning of In-kernel Software Components

For the dependable and efficient operation of a system as a whole, the lower levels of the software stack, and the operating system (OS) in particular, are especially important, as they are the foundation on top of which our applications and services are built. Fault containment is a useful strategy to improve software dependability in the presence of residual defects in deployed software components. However, in many modern OSs, there exists no isolation between different kernel components, i.e., the failure of one component can affect the whole kernel and consequently the whole system. While microkernel OSs provide user mode isolation for large parts of the OS, their improved fault isolation has historically come at the cost of performance. Despite significant improvements in modern microkernels, monolithic OSs like Linux are still prevalent in many systems. To achieve fault isolation in addition to high performance and code re-use in these systems, hybrid approaches that relocate only fractions of kernel code into user mode have been proposed. These approaches statically decide which code to isolate, neglecting dynamic properties like invocation frequencies of the targeted components. We propose to augment static code analyses with runtime profiling to achieve better estimates of dynamic properties for common case operation. We assess the impact of runtime data on the decision of what code to isolate and the impact of that decision on the performance of such "microkernelized" systems. We extend existing tools to implement automated code partitioning for existing monolithic kernel code in an integer linear programming (ILP) framework and validate our approach in a case study of two widely used Linux device drivers and a file system. The contents of this chapter are, in parts verbatim, based on material from [Sch+18b].

2.1 Overview

Modern operating system (OS) implementations either follow a monolithic or a microkernel architecture. Microkernel OSs strive to execute a bare minimum of their overall code base in privileged kernel mode [Lie]. Code that handles resource management, for instance, is separated in code that implements the actual resource allocation and deallocation *mechanism* and code that implements the resource allocation and deallocation *policy*. In microkernel OSs, only the former executes in

kernel mode, which is sufficient to maintain non-interference of processes across shared resources. Monolithic OSs, on the contrary, execute a much larger fraction of their code base in kernel mode.

Traditionally, microkernel OSs were used for applications with high reliability requirements for two reasons. First, a small kernel code base is easier to understand and analyze, as the formal verification of the seL4 microkernel system demonstrates [Kle+09; Kle+14]. Second, the effects of individual component failures at run time are limited to the respective components due to the fine-grained isolation among system components, which facilitates the implementation of sophisticated failover mechanisms (e.g., [DH12; Her+09; Hru+12]).

Despite their reliability advantages over monolithic OSs, microkernels are seldom found in mobile, desktop, or server systems, even though reliability is a key concern for the latter. The reason for the dominance of monolithic systems in these areas lies in the poor IPC performance of early microkernel implementations, which led to significant overheads in operation. Although modern microkernels, such as the L4 family, feature highly optimized IPC implementations that reduce such overheads to a negligible fraction, their adoption is still mostly limited to embedded systems.

Ironically, the reason behind the predominance of monolithic OSs in commodity systems seems to be what is generally perceived as their major drawback. They execute large and complex code bases in privileged kernel mode within a single address space. For instance, Linux 4.7 comprised almost 22 million lines of code in July 2016 [CK16]. Reliable figures are difficult to obtain for proprietary OSs, but estimates for the Windows OS family are significantly higher [Wal12].

On the one hand, this massive complexity entails severe threats to the reliability of OSs. As complex kernel code is difficult to develop and maintain, it is likely to contain software defects. Moreover, defects are likely to escape testing and other quality assurance measures since existing testing and verification techniques do not scale well to complex software. Such *residual defects* in kernel code have a high impact on system reliability if they get triggered during execution in privileged mode because there is no limit to the degree by which they can affect processes and system services. The risks of high software complexity have resulted in the proposal of small trusted/reliable computing base architectures (e.g., [ED12; Hoh+04; Kle+09; Rus81]) for systems with high security or dependability requirements.

On the other hand, a large existing code base (and developer community that maintains it) also implies that the large amount of functionality it implements can be reused at low effort. Therefore, convenience has outweighed performance as a criterion for adopting a popular monolithic commodity OS over a microkernel OS. Early approaches like SawMill Linux [Gef+00] proposed to address this problem by manually integrating parts of Linux into the Exokernel and L4 OSs. Unfortunately, porting components across OSs is not a one-time effort and requires repeated manual adjustment as the forked code branches evolve. Ganapathy et al. developed

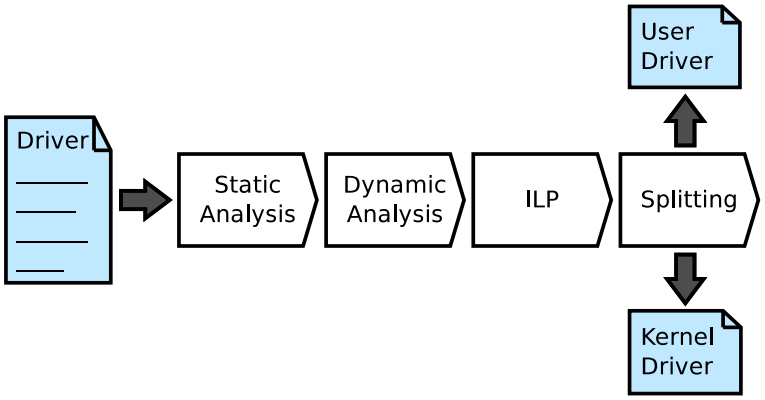


Figure 2.1: Overview of the partitioning process, exemplified for a device driver. The input is the original driver source code and the output is a split mode version of the driver that implements the isolation/performance trade-off suitable for the user's application scenario.

an approach that addresses this problem by automatically splitting the kernel code of Linux at a per function granularity into user and kernel mode portions [Gan+08]. The splitting is guided by a static set of rules that determine which code to allocate to which execution mode. While such automated splitting approaches cannot be expected to achieve all advantages of real microkernel OSs to the same degree, they still provide *better* isolation compared to a fully monolithic kernel without constraining code reuse. Unfortunately, the automated synchronization of data structures, locks, etc. between the user and kernel mode portions can entail performance overheads that exceed IPC induced overheads of microkernels by far.

Intuitively, these overheads highly depend on the kernel code *partitioning*, i.e., the decision which OS functions to execute in which mode. Moreover, the overheads depend on the system's application context. Function invocations across domains (from kernel to user mode or vice versa) entail performance overheads *per invocation*, making the performance of a partitioning dependent on dynamic system properties induced by the system's workload. The more frequent cross-domain invocations caused by a workload, the higher the overheads. This chapter addresses the central question *how to achieve a favorable partitioning that minimizes both performance overheads and the amount of code executing in kernel mode at the same time*.

Figure 2.1 gives an overview of our proposed approach. We start from the source code of some kernel component (e.g., a driver) as input and produce a split mode version as output that is tailored to the user's application scenario. First, we extract static code properties, such as the static call graph, in a static analysis phase. We then generate an instrumented component version which is used to collect the dynamic usage profile under a typical workload. We then combine the statically and dynamically obtained data to formulate and solve the kernel

component partitioning as 0-1 integer linear programming (ILP) problem. Our ILP formulation allows the fine-tuning of the trade-off between imposed overhead and amount of code that remains in the kernel. As final step, we synthesize a split mode version of the original component. The generated code for the split version is not intended for manual inspection or modification. Code maintenance, debugging, and evolution should still happen on the original code. Re-partitioning of evolved code is a simple mechanical task with our automated partitioning tool chain.

As a brief summary, this chapter presents the following contributions.

- We propose to combine static and dynamic analyses to accurately assess the performance costs that moving kernel code to user mode would cause. Our assessment is automated and works on current Linux code.
- Using the dynamically recorded cost data, we model user/kernel mode partitioning of existing kernel code as 0-1 ILP problem and use the GNU Linear Programming Kit (GLPK) [Fre] to obtain a solution that achieves the desired trade-off between performance overhead and the size of the kernel mode code portion for improved isolation.
- We validate the utility of our approach by profiling and partitioning two device drivers and a file system in a case study and demonstrate the adaptability of the obtained partitioning to user requirements.

After a discussion of related work in Section 2.2, we introduce our system model for in-kernel software components and their partitioning in Section 2.3 and detail our profiling-based partitioning approach in Section 2.4. In Section 2.5, we demonstrate its utility by applying it to Linux kernel modules and compare the obtained partitionings and their performance characteristics. Section 2.6 summarizes insights gained from the results of our study and the required implementation work. Finally, Section 2.7 concludes this chapter.

2.2 Related Work

Software partitioning, also compartmentalization or disaggregation, is an important task in iterative software development and maintenance. Surprisingly, most research in this field has focused on the design of isolation mechanisms (i.e., *how* to isolate), whereas little work covers the actual partitioning process (i.e., *what* to isolate). Software partitioning has been proposed for a number of different isolation problems.

2.2.1 Privilege Separation

Privilege separation is a mechanism to prevent privilege escalation [PFH03], i.e., the unauthorized acquisition of privileges through vulnerable programs. Privilege es-

calation vulnerabilities result from security-insensitive design that does not respect the principle of *least privilege* [SS75]. In execution environments that traditionally only support(ed) coarse grained execution privileges and access control, such as Unix and derivatives, implementing programs according to this principle has been challenging. As a consequence, a large body of legacy software does not employ newer, more fine-grained privilege separation mechanisms (e.g., [Wat+10]).

Privilege separation partitions programs into a *monitor* component, which executes privileged operations, and an unprivileged *slave* component such that vulnerabilities in the slave partition cannot lead to unauthorized execution of privileged operations. Although a large variety of mechanisms to isolate the slave from the monitor have been proposed in the literature [Col+11; Kil03; Li+14; MMH08; MWC10; Wat+10], the partitioning into suitable compartments is usually performed manually for few selected applications.

Privtrans [BS04] automates privilege separation for C programs based on user-supplied source code annotations that mark sensitive data and functions to be encapsulated by the monitor component. An automated data-flow analysis determines the monitor and slave partitions by propagating the annotations to all data and functions operating on or derived from annotated elements.

Wedge [Bit+08] extends the Linux kernel by several isolation primitives to implement privilege separation. To assist application partitioning into isolated compartments, the authors conduct dynamic binary instrumentation to derive interdependencies on shared memory between code blocks and their respective access modes from execution traces.

Jain et al. observe that capabilities in Linux are too coarse-grained to enforce the principle of least privilege for unprivileged users [Jai+14]. As a result, policies are commonly implemented in setuid-root binaries, a potential source of vulnerabilities. The authors present an extension of AppArmor which facilitates moving such policies to the kernel with minimal overhead.

Liu et al. employ combined static and dynamic analysis to automatically decompose an application into distinct compartments to protect user-defined sensitive data, such as private keys, from memory disclosure vulnerabilities [Liu+15].

2.2.2 Refactoring

Refactoring denotes the restructuring of software to improve non-functional properties without altering its functionality [Fow99]. It comprises the decomposition of monolithic software systems as well as changes in an existing modular structure. Call graphs, module dependency graphs, or data dependency graphs are commonly used to represent software structures for refactoring (e.g., [CV95; DK99; Sha+03; Ton01]). Whether nodes in such graphs should be merged, split, or remain separate is usually decided by cohesion and coupling metrics [YC79] associated

with the represented software structures, either by graph partitioning [Bav+; CV95; Sha+03], cluster analysis [LA02; LZN04; MB07; Wig97], or search based techniques [HHP02; MM06; PHY11].

2.2.3 Mobile/Cloud Partitioning

In order to enable sophisticated, computationally demanding applications to run on resource and energy constrained mobile devices, the partitioning of such applications into more and less demanding compartments has been proposed in the literature [Chu+11; MN10; Yan+13]. The former is then executed on the mobile device itself whereas the latter is executed remotely on a server infrastructure without draining the battery. Due to the dynamically changing operational conditions of mobile devices (battery strength, bandwidth, etc.), most approaches combine static and dynamic measures for partitioning, similar to the approach presented in this chapter.

2.2.4 Fault Tolerance

A large number of approaches have been proposed to isolate critical kernel code from less critical kernel extensions, as existing extension mechanisms were found to threaten system stability in case of misbehaving extensions [Cho+01; Gan05; GGP06; MN07; Pal+11; Sim03]. Similar to privilege separation, most work in this field has focused on *how* to establish isolation between the kernel and its extensions [Cas+09; Jo+10; Kan09; LeV+04; Mao+11; NB13; SBL03; SC13; Spe+06; Tan+07; Wil+08; Zho+06], but only little work considers the problem of identifying *what* to isolate for achieving improved fault tolerance at an acceptable degree of performance degradation.

Ganapathy et al. target this question in the Microdrivers approach [Gan+08] that proposes a split-mode driver architecture, which supports the automated splitting of existing Linux device drivers into user and kernel compartments. The splitting is based on static analyses of the driver code and a set of static rules for classifying functions as either performance critical or uncritical. The approach has been implemented in a tool called "Driverslicer", a plugin for the CIL source code transformation and analysis tool chain [Ker; Nec+02]. Renzelmann et al. extend Microdrivers to support Java for reimplementing the user part of split device drivers [RS09]. Butt et al. extend the Microdrivers architecture by security aspects using dynamically inferred likely data structure invariants to ensure kernel data structure integrity when data is transferred from the user part to the kernel part [But+09].

In this chapter, we demonstrate that the addition of runtime information on performance measures significantly improves the partitioning by avoiding static worst-case approximations. We use this information to state partitioning as a

0-1 ILP problem, for which we obtain an *optimal solution* with respect to a given isolation/performance trade-off.

2.3 System Model

We consider the problem of bi-partitioning a kernel *software component* S (e.g., kernel modules) into a user mode fraction S^u and a kernel mode fraction S^κ for split-mode operation to reduce kernel code size, where mode transitions occasion a cost c. We detail our software component model, cost model, and code size notion in the following subsections. This provides the foundations for stating kernel/user mode partitioning as 0-1 ILP problem in Section 2.4.

2.3.1 Software Component Model

As we target kernel code for partitioning, we assume S to be written in a procedural language like C. In procedural languages, a software component S comprises a finite set of *functions* $F(S) = \{f_i \mid i \in \mathbb{N}\}$[1]. Any function f_j can be *referenced* by any other function f_i of the same component and we denote such references by $f_i \rightsquigarrow f_j$. Our reference notion comprises direct (function calls) and indirect (passing function pointers as arguments) references [Ryd79]. Using the reference relation on functions, we obtain the *call graph* $(F(S), R(S))$, where $F(S)$ represent vertices and $R(S) = \{(a, b) \in F(S) \times F(S) \mid a \rightsquigarrow b\}$ edges of the graph.

Kernel Interactions

As allocating functions in S that heavily interact with kernel functions external to S to the user mode partition would significantly affect performance, we extend our software component model to describe such interactions. We have to consider two cases: (1) functions in S are invoked from other parts of the kernel not in S and (2) functions in S invoke kernel functions external to S. Hence, we add a *kernel node* $\mathfrak{K}$ and corresponding edges for references from and to such functions not in S to the call graph. We define the extended call graph as

$$\begin{aligned}(F'(S), R'(S)) = (&F(S) \cup \{\mathfrak{K}\},\\ &R(S) \cup \{(\mathfrak{K}, f) \mid f \in F_{entry}(S)\}\\ &\cup \{(e, \mathfrak{K}) \mid e \in F_{ext}(S)\}),\end{aligned}$$

where $F_{ext}(S) \subseteq F(S)$ is the set of functions that reference any function declared as `extern` in the program code of S, and $F_{entry}(S) \subseteq F(S)$ is the set of all functions on which the address-of operator (`&` in the C language) is used, i.e., functions

[1] We do not include 0 in $\mathbb{N}$. In cases that include 0, we use $\mathbb{N}_0$.

potentially invoked by component-external code. Note that $\mathfrak{K}$ represents any function that resides within the kernel but is external to S, including core kernel functions as well as other in-kernel software components.

Data References

When loaded into memory, S resides in a memory *address space* $A(S) = [\bot_S, \top_S]$ with lower and upper bound addresses $\bot_S, \top_S \in \mathbb{N}_0$. S's data is contained in a finite amount of memory allocations $M(S) = \{(a, l) \mid a \in A(S) \wedge l \in \mathbb{N}\}$ of that address space, where a denotes the starting address of an allocation and l the *length* of the allocated slot in bytes. No memory allocation can exceed the address space boundaries:

$$\forall (a, l) \in M(S),\ a + l \leq \top_S$$

and memory allocations within an address space are disjoint:

$$\forall (a, l), (a', l') \in M(S),\ a < a' \Rightarrow a + l < a'.$$

We denote the reference (read/write access) of a function $f \in F'(S)$ to allocated memory $m \in M(S)$ by $f \leftrightarrows m$.

Note that interactions on shared memory are implicitly covered by our data model, as we do not require component address spaces to be disjoint. We assume that shared memory across differing address spaces is mapped to the same addresses in all address spaces and that memory allocation lengths are also the same for shared memory.

Partitioning

By bi-partitioning S's extended call graph $(F'(S), R'(S))$, we obtain two disjoint sets $F(S^u)$ and $F(S^\kappa)$ of functions, where functions $f \in F(S^u)$ reside in the user and functions $f \in F(S^\kappa)$ in the kernel mode partition. Note that the kernel node $\mathfrak{K}$ is, per definitions, always assigned to $F(S^\kappa)$. Moreover, we obtain three disjoint sets of edges:

$$\begin{aligned} R(S^u) &= \{(a, b) \mid a, b \in F(S^u)\} \text{ and} \\ R(S^\kappa) &= \{(a, b) \mid a, b \in F(S^\kappa)\} \end{aligned}$$

are the sets of edges internal to the user and the kernel mode partitions, whereas

$$\begin{aligned} R_{cut}(S^u, S^\kappa) = \{&(a, b) \in R'(S) \mid \\ &(a \in F(S^u) \wedge b \in F(S^\kappa)) \\ &\vee (a \in F(S^\kappa) \wedge b \in F(S^u))\} \end{aligned}$$

is the set of edges cut by the partitioning, i.e., edges that represent inter-domain function invocations. Neither nodes nor edges are lost during partitioning. So, we define the set of all possible partitionings of a software component S as

$$\begin{aligned} P_S = \{ & (F(S^v), F(S^\kappa)) \mid \\ & F(S^v) \cap F(S^\kappa) = \emptyset \\ & \wedge F(S^v) \cup F(S^\kappa) = F'(S) \\ & \wedge R(S^v) \cup R(S^\kappa) \cup R_{cut}(S^v, S^\kappa) = R'(S) \}. \end{aligned} \tag{2.1}$$

The cost of the cut, and thereby the performance overhead of the partitioning, is then given by the sum of the weights of all edges in $R_{cut}(S^v, S^\kappa)$ and the isolation degree of a cut is expressed in terms of *size* of the S^κ partition; the smaller the kernel components the better the isolation. We detail both edge weights and size measures in the following.

2.3.2 Cost Model

To model the cost c associated with a partitioning $p \in P_S$ of a component S, we first define a weight function $w\colon R'(S) \to \mathbb{R}$ that assigns a weight to each edge of the extended call graph. The weight represents the expected overhead for invoking the corresponding reference as inter-domain function call. The associated overhead results from (a) mode switching overheads for changing the execution mode, (b) copying function parameters and return values between modes, and (c) synchronizing that part of the split component's state that is relevant to both partitions, i.e., memory locations m that are accessed from both partitions:

$$\begin{aligned} \{ m \in M(S) \mid \exists f_v \in F(S^v), f_\kappa \in F(S^\kappa) \colon \\ f_v \leftrightarrows m \wedge f_\kappa \leftrightarrows m \}. \end{aligned}$$

Points (b) and (c) both require copying data between the disjoint memory allocations $M(S^\kappa)$ and $M(S^v)$ which imposes an overhead that depends on the amount of data to copy. The overall weight for each edge is therefore computed according to Equation (2.2), where $t \in \mathbb{N}_0$ denotes the number of expected invocations of reference $r \in R'(S)$, $b\colon R'(S) \to \mathbb{R}$ denotes the average number of bytes transmitted upon a single invocation of a reference, and $c_{sys}\colon \mathbb{R} \to \mathbb{R}$ denotes the estimated time that mode switching and copying a number of bytes across partition boundaries takes on system *sys*. We detail the assessment of accurate edge weights using collected runtime data in Section 2.4.2.

$$w(r) = t \cdot c_{sys}(b(r)) \tag{2.2}$$

The cost for a partitioning $p \in P_S$ is given by $c : P_S \rightarrow \mathbb{R}$ as stated in Equation (2.3), i.e., the sum of edge weights of all cut edges. By minimizing $c(p)$, we can find a partitioning with minimal cut weight, i.e., a partitioning with minimal overhead for inter-domain function calls.

$$c(p) = \sum_{r_i \in R_{cut}(S^v, S^\kappa)} w(r_i) \tag{2.3}$$

2.3.3 Isolation Degree

All software components S that execute in kernel mode do not only operate with the highest system privileges they also share the same address space, i.e., $\forall S_i, S_j,\ A(S_i) = A(S_j)$. Hence, defective or malicious code within such components could arbitrarily alter any code or data in any other kernel components and, ultimately, in the entire system. *Isolation* prevents the unintended alteration of a software component's data or code by another software component by enforcing domain boundaries between components that, if at all, can only be crossed via well defined interfaces. Intuitively, the degree of isolation in a system is higher the more code is executing in unprivileged user mode within a separate address space, as this code cannot directly access data or functionality in the kernel. We, therefore, measure the degree of isolation by the amount of kernel code executing in user mode, i.e., the user partition *size*.

To account for partition sizes, we assign all functions in the extended call graph their source lines of code (SLOC) count as node weight with $n\colon F'(S) \rightarrow \mathbb{N}_0$. The size of a partition is then given by the sum of its node weights. As the kernel node $\Re$ represents the entirety of kernel functions external to S which, by definition, cannot be moved to the user mode partition, we define $n(\Re) = 0$ in order to include only component S in our size notion. Although the user partition size is a more intuitive measure for the isolation degree, we use the kernel partition size as a measure for *lack of isolation* in the following. The formulation of both optimization objectives, cut weight and partition size, as values to *minimize* facilitates their combination in a single cost function for optimization as we show later in Section 2.4.3. Due to the constraints on the node sets of user and kernel partition in Equation (2.1), both size measures for isolation are equivalent.

We define $s\colon P_S \rightarrow \mathbb{N}_0$ accordingly in Equation (2.4) for assigning a partitioning p its *lack of isolation* degree. A partitioning p with minimal $s(p)$ has the smallest possible amount of code residing in the kernel mode partition and, thus, the largest possible user mode partition, i.e., the highest isolation.

$$s(p) = \sum_{f_i \in F(S^\kappa)} n(f_i) \tag{2.4}$$

2.4 Runtime Data Driven Partitioning

In order to obtain an ideal partitioning of a software component with respect to a desired isolation/performance trade-off according to our system model, we need to (1) perform a static code analysis to extract the component's call graph, the node weights (SLOCs), and the sets of possible data references from its program code, (2) perform a dynamic analysis of the component to assign edge weights (expected cross-mode invocation costs) to model the impact of partitioning on our objectives, and (3) formulate our optimization objectives and constraints as ILP problem.

To implement the approach outlined in Figure 2.1 on page 23, we reuse and extend the Microdrivers framework by Ganapathy, Renzelmann et al. [Gan+08; RS09]. Originally, the framework only supported 32 bit (x86) Linux (v2.6.18), but we updated it to support contemporary 64 bit (x86_64) Linux versions (v3.14+). Our approach does not require modification of the Linux kernel beyond the component to be partitioned and, hence, is applicable to off-the-shelf kernels. Only the Microdrivers runtime and some parts of the tool chain may require updates for porting the approach to other kernel versions. We detail the individual processing steps in the following.

2.4.1 Static Analyses: Call Graph and Node Weights

We largely rely on the code analysis and transformation framework CIL [Ker; Nec+02], which uses a simplified abstract representation of C code that can be analyzed by custom plugins written in OCaml. First, we use CIL to extract the static call graph from the input software component by identifying all defined functions and all function call sites. Second, we modify the obtained call graph according to our model and introduce the kernel node $\mathfrak{K}$ and corresponding edges. We handle indirect function invocations (via pointer) by adding edges to all functions whose signatures are compatible with the invoked function pointer type. This over-approximation introduces a number of false positives, i.e., edges that do not represent possible function calls during runtime. However, we compensate for these using the recorded runtime data from our dynamic analysis (cf. Section 2.4.2). Figure 2.2 on the next page illustrates a resulting call graph, including node and edge weights.

For obtaining the node weights (n_i in Figure 2.2 on the following page), we analyze the software component's preprocessed C code and count the "physical" source lines of code (SLOC) for each function. We adopt the common SLOC notion and only include non-blank, non-comment lines. We implemented a Clang/LLVM based tool for extracting accurate SLOC counts on a per function level. We chose not to rely on CIL for this task in order not to distort the SLOC counts through CIL's code transformations, which generally increase the SLOC count disproportionately.

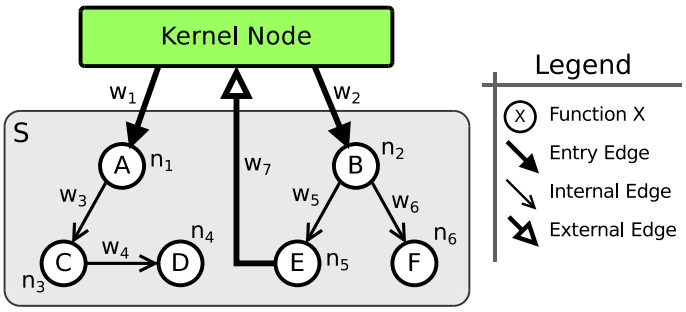


Figure 2.2: Example call graph of a kernel software component S as used for partitioning. Nodes A to F represent functions with statically determined weights n_i; edges represent possible function calls with dynamically determined weights e_i. "Kernel Node" ($\Re$ in Section 2.3.1) represents all kernel functions outside component S.

To extract the set of possible data references for each function, we reuse the marshaling (points-to) analysis of the Microdrivers framework, which is implemented as part of a CIL plugin called "Driverslicer". This is the same analysis that the Microdrivers framework employs for generating the marshaling code needed for synchronizing state between the user and kernel mode domains (see Section 2.3.2). The analysis yields an over-approximation of possible data references for each function, i.e., which data *may be reachable* from which functions. The analysis relies on programmer supplied code annotations as discussed later in Section 2.4.2. We refer the reader to the Microdrivers publications [Gan+08; RS09] for a detailed discussion of Driverslicer's marshaling analysis. We use the results of this analysis in the dynamic analysis phase for collecting runtime data as detailed in the following section.

2.4.2 Dyn. Analyses: Edge Weights & Constrained Nodes

While static analyses are useful to obtain information related to the code structure, their utility to approximate function invocation frequencies or sizes of (dynamic) data structures is limited. For instance, invocation frequencies for function calls inside a loop that depends on input data can only be sensibly estimated by a dynamic analysis; the same is true for estimating the length/size of linked data structures such as lists or buffers whose size depends on input data. We compensate for this limitation by augmenting the statically obtained structure (call graph and node weights) with data from dynamic profiling. For edge weights, relying on recorded data from dynamic profiling yields more accurate results than static over-approximations, as long as the workload used to conduct the profiling is comparable to the system load encountered during actual operation.

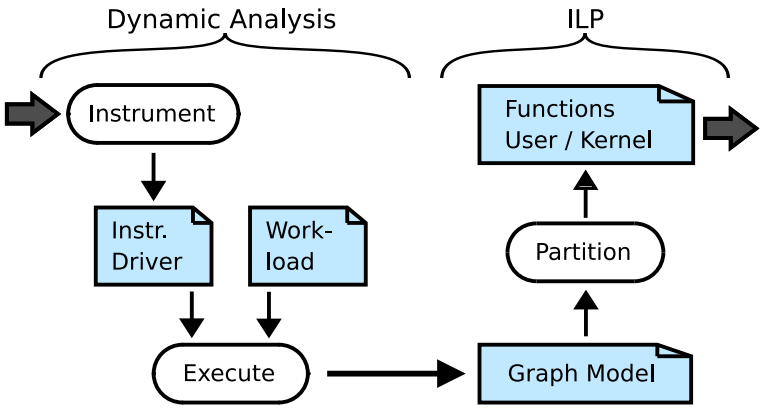


Figure 2.3: Dynamic analysis and ILP steps in the partitioning process. An instrumented version of the original kernel module is built and executed under a given workload. The collected runtime profile is used to determine the edge weights in our graph model that is used for the ILP-based partitioning, which assigns all functions to either kernel or user mode.

Edge Instrumentation

For collecting the data needed to compute the edge weights (w_i in Figure 2.2) according to our weight function $w(r)$ (cf. Equation (2.2)), we instrument the software component S and execute it to capture its dynamic behavior under a given workload. A general overview of the dynamic analysis steps is given in the left part of Figure 2.3.

We utilize the statically obtained call graph to identify relevant code locations for instrumentation. To collect data for all call graph edges that start in a node other than the kernel node $\mathfrak{K}$, i.e., edges $(f_i, f_j) \in R'(S),\ f_i \neq \mathfrak{K}$, we instrument all function call sites within S. For entry edges $(f_i, f_j) \in R'(S),\ f_i = \mathfrak{K} \wedge f_j \in F_{entry}(S)$, the call sites are external to S. Hence, we instrument the function bodies of the target functions f_j for these edges. For functions that can be invoked from within S as well as from $\mathfrak{K}$, we correct the collected entry edge data in a postprocessing step to avoid false accounting for entry edges.

We insert code at the above described code locations to record per edge: (i) the number of edge activations (function invocation frequency) (t in Equation (2.2)), (ii) the estimated data amount that would be transmitted between functions in case of an inter-domain call as arguments and return values (an addend in the calculation of b in Equation (2.2)), (iii) the estimated data amount for the synchronization of global data accessible from caller and callee (also contributing to b), and (iv) the *execution context* in which the call occurs. Information on the execution context is used to identify *constrained nodes*, i.e., nodes that cannot be moved to user mode, as discussed later in Section 2.4.2.

For the instrumentation, we employ aspect-oriented programming [Kic+97] techniques and generate the instrumentation code as separate C source code with our code generator tool, which is implemented as a CIL plugin. We use the *AspeCt-oriented C* compiler [GJ10] to insert the instrumentation code into the component during the build process. Aspect-oriented programming has the advantage that the instrumentation code is written in the same language as the code that is being instrumented, while both can be maintained as separate modules.

The inserted code implements a dynamic size estimation by walking through data structures reachable from function parameters, global variables, and return values, and summing up their sizes. Linked data structures and heap allocated structures are handled correctly by following pointers and interpreting pointer targets according to the pointed-to data type. The required data type information for this estimation technique is obtained by reusing the points-to data from the Microdrivers marshaling analysis. The analysis relies on programmer supplied annotations to fill the gaps in the data type information inherent in the C language. For instance, annotations are required to resolve `void` pointers to actual types or to specify the length of dynamically allocated buffers. Effectively, we are refining the static data type based overestimation of reachable data structures that the Microdrivers analysis provides using actual data values observed during runtime. For instance, if we observe a `NULL` pointer in a data structure, we do not consider the pointer target's data type for size estimation. The described approach is tailored for use with the Microdrivers framework. If another framework is selected to implement the split mode operation, the size estimation has to be adapted to reflect the data synchronization approach of that framework.

Using the recorded invocation frequencies and data transmission estimates from the dynamic analysis, we can derive the expected performance overhead that cutting edges in $R'(S)$ implies. As such overhead differs on different hardware platforms (and also with different frameworks used for splitting), we express the actual cost as a function c_{sys} of the amount of data to be copied. To determine c_{sys}, we implemented a kernel module for conducting measurements on the target platform, where the split mode component should ultimately execute. The module measures and records the overhead that the transfer of different data amounts causes in inter-domain function invocations. We fit a function onto the recorded data and use it to estimate the overhead for the average data sizes recorded during profiling. This completes the information required for calculating edge weights according to Equation (2.2): t is the number of observed edge activations, $c_{sys}(x)$ is the fitted platform dependent function, and $b(r)$ is the average number of bytes transmitted.

Constrained nodes

Due to the structure of commodity OSs, and in particular Linux, there are functions (nodes) that have to remain in the kernel partition. The auxiliary node $\Re$, representing all functions external to S, must remain in the kernel partition by definition. Another example are functions that may execute in interrupt context. This is an inherent limitation of the Microdrivers framework, which synchronizes between user and kernel mode via blocking functions and code running in interrupt context cannot sleep [Lov]. Consequently, we must ensure that such non-movable functions remain in the kernel partition. A number of possibilities exist to circumvent this restriction, for instance by changing the synchronization mechanisms in Microdrivers or by employing mechanisms for user mode interrupt handling, such as in VFIO [VFI16] or the Real-Time Linux patch set [RTw16]. As these only affect the achievable partitioning result and not the partitioning approach, which is the central topic of this chapter, we do not assess the impact of these options.

We denote $F_{mov}(S) \subseteq F'(S)$ as the set of movable functions and $F_{fix}(S) \subseteq F'(S)$ as the set of functions that are fixed in kernel mode. Both sets are disjoint and $F_{mov}(S) \cup F_{fix}(S) = F'(S)$. We determine $F_{fix}(S)$ using the execution context records from profiling. Every function f_i that executed in interrupt context during profiling and all functions f_j that are reachable from f_i are in $F_{fix}(S)$ (transitive closure). Note that this approach may miss some unmovable functions if they were not observed in interrupt context. Such false negatives can be mitigated in the resulting partitioning, for instance, by providing alternate code paths that allow the execution of interrupt functions within the kernel even though they were moved into the user mode partition. However, we did not encounter any such cases in our case study.

A number of kernel library functions, e.g., string functions like `strlen`, have equivalent functions in user mode libraries. These functions can be ignored for the partitioning as a version of them exists in both domains. We therefore remove them from our call graph model prior to partitioning. This is a performance optimization for the resulting split mode components.

2.4.3 Partitioning as 0-1 ILP Problem

We express our partitioning problem as 0-1 ILP problem, as illustrated in the right half in Figure 2.3. In general, stating a 0-1 ILP problem requires a set of boolean *decision variables*, a linear *objective function* on the variables to minimize or maximize, and a set of linear inequalities as *problem constraints* over the variables. Once stated as ILP problem, a linear solver can be used to find an *optimal* partitioning.

Decision Variables

We introduce the following two sets of boolean variables: $x_i, y_i \in \{0, 1\}$. For each node f_i in our call graph, a corresponding variable x_i assigns f_i to either the user or kernel mode partition as follows:

$$\forall f_i \in F'(S),\ x_i = 0 \Leftrightarrow f_i \in F(S^u) \wedge x_i = 1 \Leftrightarrow f_i \in F(S^\kappa)$$

Additionally, a variable y_i determines for each corresponding call graph edge r_i whether the edge is cut by the partitioning as follows:

$$\forall r_i \in R'(S),\ y_i = 0 \Leftrightarrow r_i \notin R_{cut} \wedge y_i = 1 \Leftrightarrow r_i \in R_{cut}.$$

Problem Constraints

Since variables x_i and y_i are boolean, we can express their relation using a boolean exclusive-or (XOR) operation $y_k = x_i \oplus x_j$, where y_k encodes if edges $r_k = (f_i, f_j)$ are cut or not and x_i, x_j represent the partition assignments of the two adjacent nodes. In order to express this relation as a linear equation system, we define four constrains for each edge as given in Equations (2.5) to (2.8). The constraints encode the boolean truth table for XOR, one equation per row in the truth table.

$$x_i + x_j - y_k \geq 0 \tag{2.5}$$
$$x_i - x_j - y_k \leq 0 \tag{2.6}$$
$$x_j - x_i - y_k \leq 0 \tag{2.7}$$
$$x_i + x_j + y_k \leq 2 \tag{2.8}$$

In addition to the XOR encoding, we need further constrains to fix non-movable functions as discussed above in the kernel partition, i.e., $\forall f_i \in F_{fix}(S),\ x_i = 0$. We achieve this by adding one additional constraint of the form given in Equation (2.9) per non-movable function f_i.

$$x_i \leq 0 \tag{2.9}$$

Objective Function

We combine the cost (Equation (2.3)) and size (Equation (2.4)) functions from Section 2.3 to a single objective function with a balance parameter $\lambda \in [0, 1]$. We compute the edge weights w_i and node weights n_i as described above for all functions and edges in $(F'(S), R'(S))$. We then reformulate our minimization objectives $c(p)$ and $s(p)$ as sums over normalized edge and node weights including decision variables as defined in Equations (2.10) and (2.11). The node and edge weights are normalized to the interval $[0, 1]$ according to Equation (2.12) which also

normalizes both equations. The normalized weights represent percentages of the overall weight present in the call graph.

$$c'(S) = \sum_{r_i \in R'(S)} ||w_i|| \cdot y_i \tag{2.10}$$

$$s'(S) = \sum_{f_i \in F'(S)} ||n_i|| \cdot x_i \tag{2.11}$$

$$||a_i|| = \frac{a_i}{\sum_{j=1}^{n} a_j} \tag{2.12}$$

Combining Equations (2.10) and (2.11) into one linear function with a balance parameter λ yields Equation (2.13), which is our final objective function for the ILP solver.

$$obj(S) = \lambda \cdot c'(S) + (1 - \lambda) \cdot s'(S) \tag{2.13}$$

λ enables the tuning of the trade-off between the expected performance overhead and the amount of code that resides in the user partition. Setting λ to a value near 1 prioritizes the minimization of the performance overhead, i.e., cut cost $c(p)$. In this case, a resulting partitioning can be expected to have a near zero cut cost, i.e., negligible performance overhead, but a large kernel partition. Setting λ to a value near 0 prioritizes the minimization of the kernel partition, i.e., SLOC count $s(p)$. A partitioning in this case can be expected to have a kernel partition as small as possible, but a high performance overhead.

2.5 Evaluation

We demonstrate the utility of our approach in a case study of three Linux kernel modules: two device drivers (`psmouse` and `8139too`) and one file system (`romfs`). For the dynamic analyses, we expose the instrumented kernel modules to throughput benchmarks and collect their runtime profiles. We derive the platform overhead functions c_{sys} for our target systems from additional measurements and use them with the obtained profiles for generating and comparing partitionings with different isolation/performance trade-offs. We highlight general insights from this process that are not limited to the scope of our case study.

2.5.1 Experimental Setup

We start by describing our experimental setup that we use for our evaluation. We conduct our experiments on two different test machine setups:

(1) a physical machine setup that we term PHY

(2) a virtual machine setup that we term VM

Both systems run Debian 8.4 (Jessy) as operating system with Linux 3.14.65 (long-term support) in a 64-bit (x86_64) configuration. Our physical machine (PHY) is equipped with a quad-core Intel i7-4790 CPU running at 3.6 GHz, 16 GiB of RAM main memory, a 500 GB SSD, and a 200 GB HDD. Our virtual system (VM) emulates a dual-core CPU and 1 GiB of RAM main memory, with the virtual CPU cores being pinned to physical cores on the host machine and the guest RAM being locked on the host. As virtualization platform, we employ QEMU/KVM 2.1.2 using the just described physical machine as host. Note that we use the VM setup only for `psmouse` experiments as we rely on QEMU's ability to emulate mouse events. The HDD is used for `romfs` experiments.

Test Module Selection

To demonstrate the applicability of our approach to general in-kernel components, we select kernel modules that utilize distinct kernel interfaces and exhibit different runtime characteristics for our evaluation. Table 2.1 on the facing page lists the kernel modules we selected for that purpose. `8139too` is the driver for RealTek RTL-8139 Fast Ethernet NICs, which executes mostly in interrupt context and interacts with the kernel's networking subsystem. `psmouse` is the driver for most serial pointing devices (mouse, trackpads), which executes largely in interrupt context, has complex device detection and configuration logic, and interacts with the kernel's serial I/O and input subsystems. `romfs` is a read-only file system used for embedded systems; it does not execute in interrupt context and interacts with the kernel's virtual file system and block I/O infrastructure.

Table 2.1 on the next page reports static size metrics for the selected test modules. The *SLOC* columns list the physical source lines of code[2] before and after code preprocessing (as part of the build process). The *Functions* columns list the number $|F(S)|$ of functions implemented in the module (*All*) and the number $|F_{entry}(S)|$ of entry point functions (*Entry*). Column *Extern* lists the number of external functions referenced, *Lib* lists the number of library functions that exist in both kernel and user mode, and *Calls* lists the number $|R'(S)|$ of calls (references). Judging by the presented numbers, `8139too` is the most complex of the modules with more code, more functions, and a larger interface with the kernel, which results in a higher coupling with other kernel subsystems than the other modules. The relatively high number of entry functions for `psmouse` is due the driver's heavy usage of function pointers rather than a large exported interface (see Section 2.3.1 on page 27).

Workload Selection

As workloads, we apply throughput benchmarks with a duration of 60 s to all test modules. For `8139too`, we use `netperf` 2.6 in *TCP_STREAM* mode measuring the

[2] generated using David A. Wheeler's SLOCCount

Table 2.1: Overview of the selected test modules. SLOC columns report original and preprocessed line numbers. Function columns list the number of overall and entry functions. The remaining columns list the number of external & library functions and call sites.

	SLOC		Functions				
Module	Orig	PreProc	All	Entry	Extern	Lib	Calls
8139too	2087	38 042	123	35	69	19	378
psmouse	1390	20 779	59	26	42	2	214
romfs	927	27 448	42	14	25	4	96

network throughput. For `psmouse`, we use QEMU's monitor and control interface (QMP) to generate mouse move events measuring the event throughput. For `romfs`, we use `fio` 2.2.9 to perform file read tests measuring read throughput. All workloads contain the module loading/unloading steps and all initialization/cleanup operations, such as `mount`/`umount` for `romfs` and `ifup`/`ifdown` for `8139too`.

2.5.2 Instrumentation & Profiling

We instrument all modules with our aspect-oriented instrumentation tool and execute them in our test systems using the aforementioned workloads. The instrumented modules are only used to collect profiling data; they are removed from the system once profiling is complete. We repeat the profiling runs 50 times for each module, rebooting the systems before each run to avoid interferences between runs. In addition to profiling runs, we also perform 50 runs with the non-instrumented module as a baseline to determine the runtime overhead incurred by the instrumentation and the split mode operation.

Instrumentation Overhead

In terms of binary module size, the instrumented module versions are about 12 to 42 times larger than the original ones. This is due to the aspect-oriented instrumentation approach, which produces additional C code for each function call site. We report performance measurements for the instrumented and the original module versions in Table 2.4 on page 46 (first two rows) together with our overall results. As apparent from columns *Throughput*, *Ifup/Ifdown* and *Mount/Unmount*, the instrumentation does not impact throughputs or init/cleanup times. Module load/unload times increase slightly for some modules, with a maximum increase of factor 3.6 for loading `8139too`. We therefore conclude:

Table 2.2: Runtime profile overview. The first four columns list the number of activated function nodes and call edges (absolute and relative); the last column reports the relative amount of movable functions.

	Activations		Rel. Coverage		
S	$F(S)$	$R'(S)$	$F(S)$	$R'(S)$	$F_{mov}(S)/F(S)$
8139too	82	201	66.7 %	53.2 %	65.9 %
psmouse	36	81	61.0 %	37.9 %	83.1 %
romfs	36	84	85.7 %	87.5 %	83.3 %

Aspect-orientation provides a modular way to implement source code instrumentation on the abstraction level of the targeted programming language with overheads small enough to allow production usage.

Runtime Profiles

Table 2.2 gives an overview of the observed runtime profiles. The *Activations* columns list the number of functions and references that our workload activated, whereas the *Rel. Coverage* columns report the relative amount of activations. The last column reports the percentage of functions that our partitioner may move to the user mode partition, i.e., the number $|F_{mov}(S)|$ of nodes for which no constraints apply (cf. Section 2.4.2). Our `romfs` workload achieves the highest coverage as this module only contains the essentials for reading from the file system. The percentage of constrained nodes is lowest here as `romfs` does not execute in interrupt context and only needs a few functions fixed in the kernel to ensure correct operation in split mode. The relatively low percentage of activated calls in `psmouse` is due to the usage of function pointers as well as the high amount of device specific detection and configuration logic, most of which is not needed for our emulated QEMU mouse device. The few unmovable functions in this module execute in interrupt context. `8139too` has the lowest fraction of movable functions as this driver primarily executes in interrupt context for network package handling. In summary, there is significant potential for moving functions to user mode for `psmouse` and `romfs` since only a small fraction of functions needs to be fixed in the kernel. The potential for moving many functions without severe performance implications is particularly high for `psmouse` and `8139too` in the given usage scenario as functions without activations can be moved to user mode without affecting the performance under the respective common case usage.

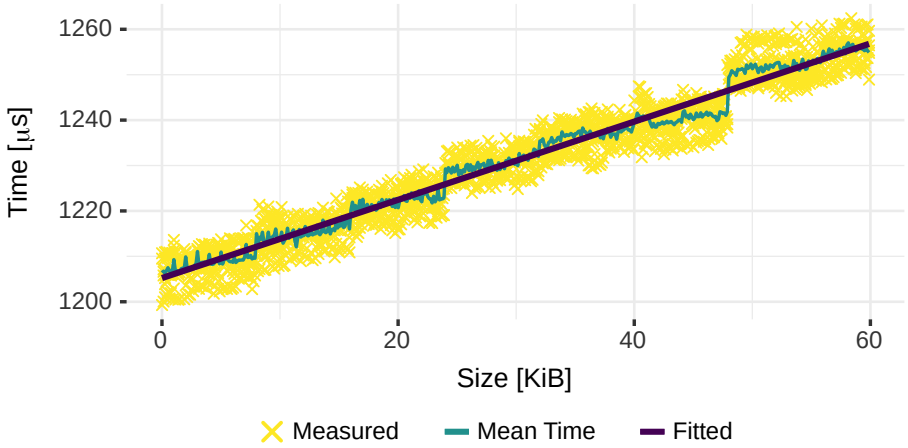


Figure 2.4: Platform overhead c_{sys} for data sizes from 0 to 60 KiB measured for our physical machine setup.

2.5.3 Estimation of the Platform Overhead

We estimate the platform overhead function c_{sys} for both our setups (PHY & VM) using a split mode test module (cf. Section 2.4.2) that measures the time needed for inter-domain function invocations with data of increasing sizes up to 60 KiB in 128 B steps. All data sizes recorded during profiling fall into this range. For each size step, we measure 1000 inter-domain calls and use their average time as the result for each step. We repeat the overall measurement process 10 times and fit a linear function onto the average measurements as we are interested in getting a mean overhead estimation.

Figure 2.4 illustrates the results for our PHY system and Figure 2.5 on the following page for our VM system. The horizontal axis shows the size in KiB whereas the vertical axis shows the measured time in microseconds. The fitted linear function for our PHY setup can be written as $c_{sys}(b) = 1205.2 + 0.00084 \cdot b$ (coefficient of determination $r^2 = 0.98$), and the function for our VM setup as $c_{sys}(b) = 1259.7 + 0.00082 \cdot b$ (with $r^2 = 0.97$). For both systems, there is a considerable static overhead of about 1205 µs for PHY and about 1260 µs for VM associated with every inter-domain function invocation. The actual data transfer entails a much smaller overhead of about 0.8 µs per 1 KiB. In earlier experiments on Linux 3.14.40, we observed a static overhead of 1214 µs for PHY, indicating that c_{sys} results may be reused across different revisions of the same kernel.

2.5.4 Partitioning Results

We use the GLPK IP/MIP solver v4.55 [Fre] for partitioning. We generate 13 partitionings per module using different λ values to investigate the effect on the

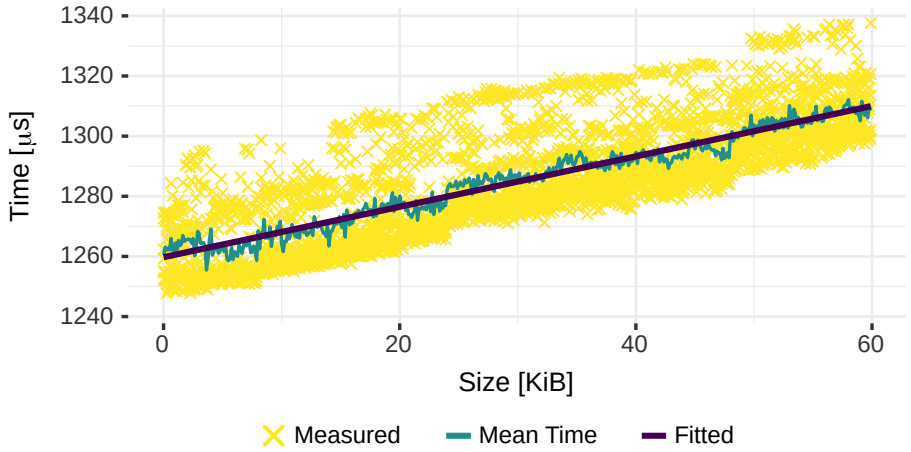


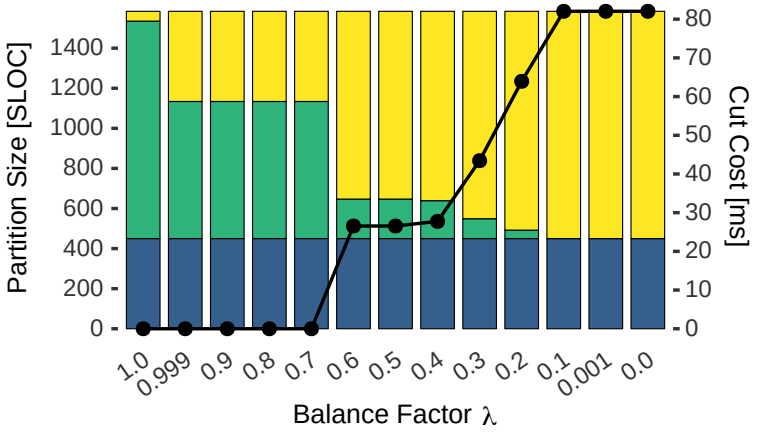
Figure 2.5: Platform overhead c_{sys} for virtual machine.

resulting partitions. For `8139too`, the solver needs on average about 1.3 MiB of RAM with 362 decision variables (after problem preprocessing). For `psmouse`, it uses about 0.8 MiB with 241 decision variables; `romfs` needs 0.4 MiB with 128 decision variables. The solver runtimes are negligible as they are reported with 0.0 s in all runs. These numbers demonstrate that, although 0-1 ILP problems are generally NP-complete, our optimization-based partitioning approach is suitable for realistic problem sizes.

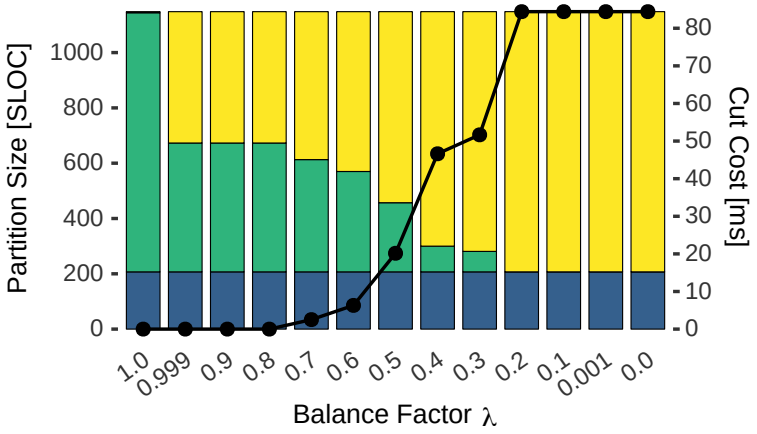
> *If stated as 0-1 ILP problem, optimal partitioning of real-world kernel software components can be achieved with modest computational overhead.*

Figures 2.6a to 2.6c on the next page illustrate the sizes ($s(p)$) and cut costs ($c(p)$) of the generated partitions. The horizontal axes display the used λ values (being identical for all modules) whereas the left vertical axes show the sizes for kernel and user partitions (in SLOC); the right axes show the cut costs (in time units). Note that the figures use different scales on the vertical axes, with Figure 2.6c using seconds and the others using milliseconds. Moreover, Table 2.3 reports the exact numbers for partition sizes and estimated cut costs for all three modules.

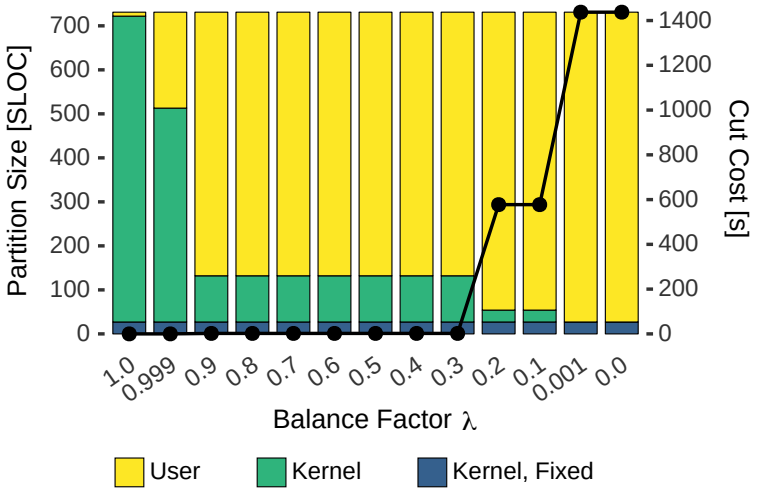
Obviously, the amount of fixed kernel code does not change with varying λ, i.e., the minimal kernel partition size is bounded by the amount of non-movable (interrupt) functions. Nonetheless, the overall kernel partition sizes decrease with decreasing λ as the minimization of $s(p)$ gains priority. The cost, however, increases with decreasing kernel partition size as more and heavier graph edges are cut with less priority on the minimization of $c(p)$. As λ approaches 0.0, a high cost must be payed even for small decreases in the kernel partition size.



(a) 8139too



(b) psmouse



(c) romfs

Figure 2.6: Development of partition sizes (left axis) and cut costs (right axis) for varying λ values (decreasing left to right) for our three test modules. The kernel partition size decreases with decreasing λ whereas cut costs increase, i.e., the more code is moved to the user partition, the higher the performance impact.

Table 2.3: Partitioning results for `8139too`, `psmouse`, and `romfs`. Only distinct partitions are reported along with the producing λ, the kernel and user partition sizes (SLOC) and estimated cost of the respective partitioning.

8139too			
λ	Kern	User	Cost (ms)
1.0	1535	49	0.0
0.7	1134	450	0.0
0.5	647	937	26.6
0.4	639	945	27.8
0.3	549	1035	43.4
0.2	492	1092	63.9
0.0	450	1134	82.0

psmouse			
λ	Kern	User	Cost (ms)
1.0	1144	4	0.0
0.8	673	475	0.0
0.7	613	535	2.5
0.6	570	578	6.3
0.5	457	691	20.2
0.4	300	848	46.6
0.3	281	867	51.7
0.0	207	941	84.4

romfs			
λ	Kern	User	Cost (s)
1.0	722	9	0.0
0.999	513	218	0.024
0.3	132	599	2.1
0.1	54	677	577.5
0.0	27	704	1437.4

If interrupt handling in device drivers was revised to allow for execution in process context, larger portions of their code could be isolated as user mode processes.

For all modules, the kernel partition is smallest at $\lambda = 0$ with highest cut cost. For $\lambda = 1$, the opposite is the case as the kernel partition size is highest and the cut cost is with a value of 0 the lowest. Decreasing λ from 1.0 to 0.999 allows the solver to find a partitioning that not only has a low cut cost, but also a larger user mode partition. This effect occurs as the solver solely minimizes the cut cost at $\lambda = 1$ without taking any node weights into account, i.e., *any* partitioning having minimal cost (in our scenario 0) is optimal for the solver, irrespective of the SLOC counts left in either partition. This is also the reason why even with $\lambda = 1$, we still have some code fractions left in the user partition. Putting a little effort into node weight minimization, however, is enough for the solver to move all "cheap" nodes. In other words, all nodes that the solver can move "for free" under a given workload are actually moved to the user partition. This gives the benefit of a higher isolation, while a performance overhead must be paid only in rare occasions that are outside

the common case usage. Therefore, we recommend to not select $\lambda = 1$ but close to 1. A similar effect does not occur when we increase λ from 0.0 to 0.001 as there is no way to reduce cut costs without increasing the kernel partition size. Note that not all generated partitions are distinct as different (neighboring) λ values may result in the same partitioning. This is due to the node and edge weights being discrete (a function can only be moved as a whole). Our partitioner produces 7 distinct partitionings for `8139too`, 8 for `psmouse`, and 5 for `romfs` (cf. Table 2.3 on the facing page).

For a known usage profile, significant portions of kernel software components can be isolated at near zero performance overhead in the common case.

Although `romfs` appears to be the simplest module from the static metrics presented in Table 2.1 on page 39, we expect especially large overheads as the cut costs illustrated in Figure 2.6c on page 43 are very high compared to the other two test modules. This is due to the nature of `romfs`, which moves large data chunks with high call frequencies between disk and memory. Even the partitioning with $\lambda = 0.9$ already has a cut cost of about 2.1 s. This effect is due to the edge weight normalization (see Equation (2.12)), which is applied to formulate the overall minimization problem for the solver. Workloads that lead to extreme hot spots in terms of call frequencies and/or data amount in the runtime profile require finer grained λ variations around 1.0 if the maximum size user partition with zero cost should be found, since the hot spots in the profile dominate the partitioning cost.

Information on a software component's dynamic usage profile is essential for an accurate cost estimation.

After automatically generating a spectrum of partitionings with different λ values using our approach, a system administrator can select a partitioning with the performance/isolation trade-off that best fits the requirements of the intended application scenario. Choosing the lowest λ value that meets required execution latencies, for instance, yields best effort reliability.

2.5.5 Split Mode Modules

We synthesize and build split mode modules for all distinct partitionings that we generated and expose them to our workloads for timing and throughput measurements. Table 2.4 on the next page reports the results. We highlight especially interesting numbers in bold face.

Overall, split modules with a cut cost of zero do not show different performance compared to the original modules except for slightly increased loading

Table 2.4: Performance measurements for the different versions of `8139too`, `psmouse`, and `romfs`. The reported results are averages of 50 experiment runs (standard deviation in brackets). Columns Load, Unload, Ifup, Ifdown, Mount, and Unmount report the durations of the respective operations in milliseconds. TP columns list workload throughputs. Interesting data points are highlighted in bold.

`8139too`					
Version	Load	Unload	Ifup	Ifdown	TP (Mbit/s)
orig	5.0 (3.0)	45.6 (12.4)	44.5 (1.5)	7.6 (0.1)	94.1 (0.0)
instrumented	17.8 (4.9)	44.2 (12.9)	44.6 (1.6)	7.6 (0.1)	94.1 (0.1)
split, $\lambda = 1.0$	9.9 (0.1)	45.8 (13.7)	44.5 (1.4)	7.6 (0.1)	94.1 (0.0)
split, $\lambda = 0.7$	14.0 (0.1)	48.4 (13.7)	44.8 (1.5)	7.6 (0.1)	94.2 (0.0)
split, $\lambda = 0.5$	**34.8 (0.3)**	47.8 (12.5)	**52.2 (1.4)**	7.5 (0.1)	94.1 (0.0)
split, $\lambda = 0.4$	34.8 (0.3)	42.8 (11.6)	52.4 (1.2)	7.6 (0.1)	94.2 (0.0)
split, $\lambda = 0.3$	35.0 (0.2)	47.1 (12.2)	59.7 (1.2)	18.8 (0.2)	94.1 (0.0)
split, $\lambda = 0.2$	39.2 (0.3)	72.7 (12.0)	59.3 (1.5)	18.7 (0.2)	94.2 (0.0)
split, $\lambda = 0.0$	**46.8 (1.4)**	**73.5 (14.2)**	**64.2 (1.4)**	**23.3 (0.2)**	94.1 (0.2)

`psmouse`			
Version	Load	Unload	TP (Events/s)
orig	3.5 (1.5)	57.8 (23.1)	1060.3 (13.0)
instrumented	4.4 (1.3)	67.1 (19.4)	1064.2 (16.7)
split, $\lambda = 1.0$	5.6 (2.4)	62.6 (21.6)	1059.8 (9.7)
split, $\lambda = 0.8$	5.7 (1.8)	60.8 (24.4)	1057.7 (11.1)
split, $\lambda = 0.7$	5.5 (2.4)	63.6 (21.4)	1060.6 (9.4)
split, $\lambda = 0.6$	5.1 (1.6)	59.5 (23.5)	1056.6 (7.9)
split, $\lambda = 0.5$	5.4 (1.7)	62.8 (24.9)	1056.1 (10.4)
split, $\lambda = 0.4$	**1545.3 (22.6)**	63.8 (22.1)	1057.1 (10.7)
split, $\lambda = 0.3$	**1540.0 (19.3)**	65.1 (22.4)	1059.9 (7.8)
split, $\lambda = 0.0$	**1537.8 (17.1)**	**114.5 (30.4)**	1060.1 (8.5)

`romfs`					
Version	Load	Unload	Mount	Unmount	TP (MiB/s)
orig	2.8 (1.9)	39.6 (10.2)	1.1 (1.4)	101.7 (8.1)	33.76 (0.72)
instrumented	5.7 (3.3)	36.9 (8.3)	1.1 (1.2)	100.3 (9.0)	33.62 (0.59)
split, $\lambda = 1.0$	5.3 (0.1)	35.7 (11.2)	1.2 (1.6)	99.4 (9.3)	33.60 (0.53)
split, $\lambda = 0.999$	**10.5 (0.2)**	**44.2 (11.1)**	**22.4 (1.1)**	103.8 (11.8)	33.69 (0.55)
split, $\lambda = 0.3$	11.4 (0.3)	44.0 (10.2)	37.7 (1.5)	**187.1 (11.2)**	33.73 (0.62)
split, $\lambda = 0.1$	11.9 (0.2)	45.2 (9.2)	37.6 (1.2)	**126.3 (9.5)**	**0.288 (0.00)**
split, $\lambda = 0.0$	11.9 (0.3)	**46.3 (12.1)**	**41.0 (0.8)**	**119.4 (9.3)**	**0.065 (0.00)**

times caused by the initialization of the Microdrivers runtime. The measured throughputs for the two interrupt heavy drivers (`8139too` and `psmouse`) remain stable as interrupt routines are not touched due to earlier discussed Microdrivers limitations. As soon as the estimated cut costs increase beyond zero, we observe a modest impact on operation latencies for `8139too` and `romfs`. Load times increase for both modules as well as `mount` and `unload` times for `romfs` and `ifup` time for `8139too`. The observed increases are due to the assignment of module and device initialization/configuration functions to user mode.

For `psmouse`, an increase in times becomes apparent only in later partitionings: starting with $\lambda = 0.4$, the module load times increase to 1.5 s as all the device detection and initialization logic gets moved to user mode. Although our estimated costs also make a jump for this partitioning, it is far smaller than the measured overhead. We attribute this anomaly to side effects that our model does not account for. `psmouse` initialization logic causes additional interrupts that interfere with the user mode process executing the mouse logic, which leads to more context switches and wait times for the user mode process.

All modules exhibit the largest performance decrease when the cut cost is highest and the user mode partition is largest. The measured time and performance impacts for `8139too` and `psmouse` are not prohibitively high for use in production. This is consistent with the estimated cut costs that remain below 100 ms. In contrast, `romfs` suffers from a significant decrease of two orders of magnitude in throughput starting from $\lambda = 0.1$ as the function that transfers contents between disk and memory (`romfs_readpage`) is moved to user mode. This is expected as the estimated cut cost becomes exceptionally high for large user mode partitions with about 24 min for $\lambda = 0.0$. The decrease in `umount` times between splits with $\lambda = 0.3$ and $\lambda = 0.1$ is a side effect of the observed throughput decrease. During un-mounting, `romfs` cleans up per-file i-node data structures using a function that is moved to user mode starting at $\lambda = 0.3$. Due to the lower throughput, fewer files are read as part of our fixed duration workload. Hence, fewer i-nodes need to be cleaned up and `umount` needs less time.

2.5.6 Reliability of Split Mode Modules

To assess the reliability gain of split mode modules, we conduct both a code analysis and fault injection experiments. As memory safety bugs constitute an important class of program bugs in C code, we focus on potentially invalid memory accesses via pointers. We analyzed the source code of our test modules to identify all code locations where pointers are dereferenced. In case of corrupted pointer values, dereferences can lead to invalid memory accesses, which, depending on the accessed memory location and whether it is a read or write access, can crash the kernel. Figure 2.7 on the following page illustrates the relative amount of pointer

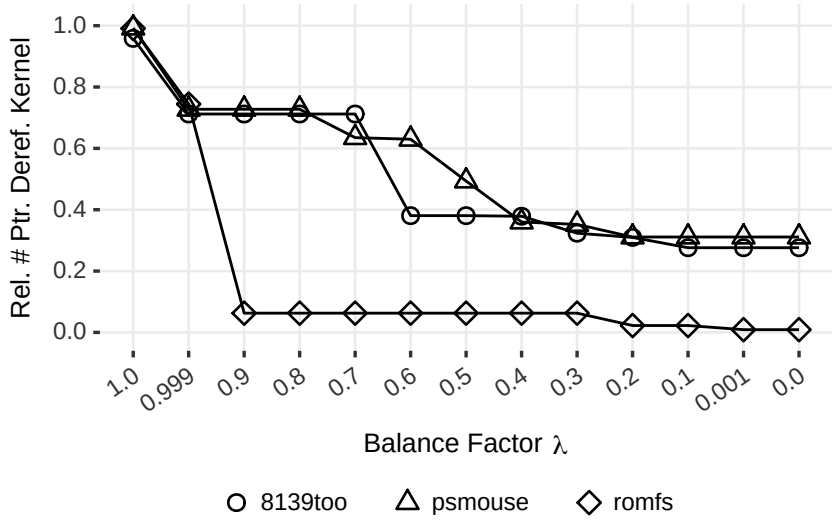


Figure 2.7: Relative amount of pointer dereferences in the kernel mode partition across different λ values. With decreasing λ, the size of the kernel mode partition decreases (left to right) along with the amount of potentially dangerous pointer dereferences.

dereferences that are left in the kernel mode partition with decreasing kernel mode size (decreasing λ). The smaller the kernel partition, the fewer potentially dangerous pointer dereferences are left inside the kernel.

Overall, we found 507 dereferences in `8139too`, 389 in `psmouse`, and 223 in `romfs`. For the smallest kernel mode partition, the amount of dereferences falls below 31 % for `8139too` and `psmouse` and for `romfs` even below 1 %. The dereferences remaining in the kernel partition reside in non-movable functions. However, at least 50 % of dereferences can be removed from the kernel partition at moderate cost (see $\lambda = 0.5$) for all three modules.

To validate that invalid memory accesses are indeed a problem when they occur inside the kernel but can be tolerated in the user partition, we conduct fault injection experiments in which we inject `NULL` pointer values into the previously identified pointer dereferences via code mutation, a fault type that kernel code is particularly prone to [Pal+11]. We randomly selected 50 mutants per module, compiled them and executed them with our workload for all previously generated partitionings. With a total of 20 distinct splits across all mutants, this sums up to a total of 1000 experiments. Across all experiments, 46 % of injected faults got activated for `romfs`, 78 % for `psmouse`, and 70 % for `8139too`. In all experiments with activated faults, the kernel reacted with an Oops and required rebooting if the invalid memory access resided in the kernel partition. However, if the invalid access resided in the user partition, the user mode driver process reacted with a segmentation fault, leaving the rest of the system unaffected. When the kernel partition size was largest ($\lambda = 1.0$), we observed an Oops in 100 % of cases. However, if the kernel partition

size was smallest ($\lambda = 0.0$), we observed Oopses only in 44 % of cases for `psmouse`, 6 % for `8139too`, and 9 % `romfs`. We conclude that the more code we move into the user partition the more potential invalid memory accesses can be isolated in the user mode driver process, thereby improving system reliability.

2.6 Discussion

Our evaluation demonstrates that kernel components can be partitioned into user and kernel compartments based on data recorded from runtime profiling while allowing for a user-defined trade-off between isolation and run-time overhead. For `romfs`, the cost for a minimal kernel component is prohibitively high, but other λ values yield usable partitionings with overheads corresponding to the amount of kernel functionality isolated. In the following, we summarize issues, insights, and practical considerations from our implementation.

A basic assumption of the used Microdrivers framework is that *data accesses are properly synchronized in the original kernel module using locking primitives*, so that shared state in the split module needs to be synchronized only upon the start and end of inter-domain function invocations and whenever a locking operation is performed. In reality, however, locks are often avoided for performance reasons, especially for small, frequently updated data fields such as the `flags` field in the `page` struct, which the Linux kernel uses for page management. Here, atomic access operations are commonly used. Atomic operations need special handling in split mode modules as the accessed data must be synchronized immediately upon access. For this reason, we left all accesses to certain fields of the `page` struct in the kernel. For instance, the `romfs_readpage` function only started working in user mode after we ensured that all page status bit accesses were left in the kernel, which reads/writes these fields concurrently using CPU-specific atomic memory operations.

State synchronization in the presence of interrupts has also proven challenging. Especially during device initialization and configuration, drivers issue device commands that may result in immediate interrupts, i.e., the driver code interrupts itself. As these commands are not automatically identified, the Microdrivers runtime is unaware of the resulting control flow redirection to the interrupt service routine. Therefore, no data synchronization is performed before the interrupt-causing operation is executed and, hence, the interrupt routine and user mode function may operate on inconsistent copies of shared data. We encountered this issue with both `8139too` and `psmouse`. Device configuration from user mode only worked after we added additional synchronization points and left certain operations in the kernel.

While we do not consider security as a partitioning goal, we note that the presented approach does not harm security: In a properly configured system, the interface between the kernel and user components of a split mode driver is not

accessible to unprivileged users. Consequently, the attack surface remains the same as for the original driver. Moreover, moving vulnerable code from the kernel to user space can reduce the severity of vulnerabilities. The user component can also benefit from hardening and mitigation techniques that may not be available or feasible in the kernel. A second consequence of our decision to not open the split mode driver's cross-domain interface to other users is that it cannot be reused, for instance by other drivers or user mode programs. This restriction is intended, as the generated interfaces are highly customized for a specific partitioning of a specific kernel component and any reuse beyond that use case bears a risk of misuse with fatal consequences.

2.7 Conclusion

Although microkernel OSs provide better isolation than monolithic OSs and modern implementations no longer suffer from the poor IPC performance of their ancestors, monolithic OSs still dominate the commodity desktop, mobile and server markets because of legacy code reuse and user familiarity. In order to benefit from both the existing code bases of monolithic systems and the design advantages of microkernel architectures, approaches to move kernel code portions of monolithic OSs into user mode have been proposed. While these approaches provide the mechanisms for split mode user/kernel operation of monolithic kernel code, they do not provide guidance on *what* code to execute in which mode. To this end, we propose a partitioning approach that combines static and dynamic analyses to assess the impact of kernel code partitioning decisions on both the degree of isolation and the expected performance overheads. Using collected data from profiling runs, we derive solutions that are *optimal* with respect to a user-defined isolation/performance prioritization.

We implement the approach for Microdrivers, an automated code partitioning framework for Linux kernel code, and demonstrate its utility in a case study of two widely used device drivers and a file system. Our results show that augmenting static analyses with data obtained from dynamic analyses allows the estimation of the performance impact and, therefore, the feasibility of a whole spectrum of possible partitionings for production usage even before a split mode version is synthesized.

For future work, we plan to address the shortcomings of existing code partitioning tools encountered during the implementation of our profiling-based partitioning and add support for atomic access operations and user mode interrupt handling. Furthermore, we plan to investigate more efficient alternatives for the complex function wrapping and parameter marshaling of Microdrivers to improve performance and the amount of movable functions.

3 Accelerating Software Fault Injections

In Chapter 2, we were concerned with the isolation of faults or defects that may be activated once software is deployed. However, testing activities during software development are essential for identifying potential issues even before deployment. Software Fault Injection (SFI) is a widely used dynamic testing technique to experimentally assess the dependability of software systems. To allow for a comprehensive view on the dependability of an increasingly complex software, SFI typically requires large numbers of experiments with potentially long execution times, which overall leads to long test latencies. To handle the increasing complexity, we propose (1) to exploit the parallel hardware resources, which have become available in virtually all desktop and server systems, for concurrent test execution and (2) to avoid redundant work between multiple executions. To that end, we investigate the efficiency and feasibility of conducting OS-level FI experiments in parallel separate virtual machines (VMs) using our PAIN (PArallel fault INjection) SFI framework. Moreover, leveraging the lessons learned from our PAIN experiences, we propose FastFI, a novel SFI framework for efficient parallel SFI experiments at higher levels of the software stack. Aiming at software levels above the OS level, FastFI is suitable for SFI on system libraries as well as applications and executes experiments in processes rather than VMs. The idea of parallel execution of experiments underlies the assumption that individual experiments do not interfere with each other in ways that change or even invalidate the experimental results. Consequently, in addition to providing increased SFI throughput, we investigate if this assumption is justified and analyze the trade-off between result accuracy and throughput increase for parallel SFI. We conduct our experiments using real-world systems and applications. For PAIN, we conduct SFI experiments in an Android OS scenario. For FastFI, we conduct experiments with PARSEC applications. The contents of this chapter are, in parts verbatim, based on material from [Sch+18a] and [Win+15b].

3.1 Overview

Modern software stacks are increasingly complex, due to the increasingly sophisticated application scenarios they are used in. To cope with this increase in

complexity, many software projects re-use existing "off-the-shelf" software components. While software re-use is cost-effective, it can pose a risk for system reliability, as even correct software can malfunction if it is used in a different context than originally anticipated. A prominent example of such a problem was the Ariane 5 incident. The inertial reference system that was safe for the Ariane 4 launcher turned out to be unsafe for Ariane 5, which exhibited a higher horizontal acceleration during the first 40 seconds after lift-off [Lio+96]. To test whether software faults in some part of the software stack are critical to its overall dependability, software fault injection (SFI) [CN13; DM06; Voa+97] is a widely used testing method.

SFI creates a number of faulty software versions, executes them, and monitors their effects on the execution environment. How SFI generates faults is commonly specified in terms of code patterns that are referred to as fault models (e.g., [CB89; KIT93; Nat+13; Rod+99]) or mutation operators (mostly in the mutation testing community, e.g., [Bud+80; DO91; JH11]). As these patterns can be applied more often for larger code bases, more complex software yields higher numbers of faulty versions and higher numbers of faulty versions result in longer SFI test latencies. Hence, SFI-based dependability assessments require increasing numbers of experiments to provide a comprehensive picture, with studies reporting tremendous numbers of experiments [Arl+02; Di +12; KD00; Nat+13]. The problem of exploding experiment numbers is reinforced by the emergence of simultaneous fault injections where multiple faults are combined and injected at once. Recent studies have shown that some dependability issues can only be discovered by the combination of multiple faults [Gun+11; JGS11; Lan+14; Win+13]. As a consequence, there is a combinatorial explosion in the number of experiments, which poses a considerable challenge in practice.

For coping with these high experiment numbers, two strategies are often adopted. The first one being the reduction of experiments by selective execution, for instance, by employing search or downsampling approaches based on heuristics [JGS11; JH08; Nat+13; SAM08]. Such a reduction is obviously unsound, as it may miss relevant (i.e., failing) tests. The second strategy is to utilize the computational power of modern parallel hardware to execute multiple experiments at the same time on the same host machine (e.g., [Dua+06; Las05; OU10]). Although parallelization could be considered the less elegant approach, it has the advantage of being generally applicable and does not require domain-specific knowledge as is usually the case for downsampling techniques. Hence, it appears to be a promising solution that can still be combined with downsampling or search strategies.

In the rest of this chapter, we first conduct parallel SFI experiments on the OS level, using the widely adopted Android OS, in a full stack virtual environment with strong isolation between individual experiments. We rely on our PAIN framework and analyze if experiment throughput can be increased by this parallelization strategy. We furthermore assess whether the parallel execution of the individual

experiments affects the obtained experiment results and give guidance on how to design such full stack parallel experiments to achieve both high experiment throughput and result accuracy. We detail our methodology and setup in Section 3.2 and continue in Section 3.3 with the presentation of our experiment results and their analysis.

We then present FastFI, our approach for the parallel execution of SFI experiments for software above the OS layer that is based on the experiences and conclusions from our PAIN experiments. In contrast to PAIN, FastFI uses lightweight isolation between experiments and relies on parallel processes rather than VMs for experiment execution. Beyond, the parallel execution of experiments, FastFI employs further techniques to reduce the number executed experiments by not even starting experiments for faults that cannot be reached during execution. FastFI also tries to avoid re-executing redundant work within the targeted application. In Section 3.4, we present the design and implementation of FastFI and continue with its evaluation in Section 3.5 by applying it to PARSEC applications. Section 3.6 discusses work related to both our PAIN experiments as well as our FastFI design. Finally, Section 3.7 concludes this chapter.

3.2 PAIN Experiments

In this section, we present the design of our PAIN experiments for the assessment of the feasibility of conducting parallel OS-level SFI experiments for increased throughput while maintaining result validity.

3.2.1 Overview

The attempt to parallelize SFI experiments relies on the assumption that the parallel execution of experiments does not impact the results. We hypothesize that this assumption is all but trivial. Even if great care is taken to avoid obvious sources of interferences between experiments, e.g., isolating them in VMs, there are many subtle factors related to timing and resource contention than may influence the target system in unexpected ways, especially if the target system is by itself a complex system such as an OS, thereby adversely influencing obtained results. For instance in the domain of embedded, real-time, and systems software, studies have shown that faults often show time-sensitive and non-deterministic behavior [Arl+02; Cot+13a].

This is why we propose the PAIN (PArallel fault INjection) SFI framework to conduct efficient and accurate parallel SFI experiments. The PAIN software framework that we developed for running our experiments, which consists of about 14 thousand source lines of code[1], is publicly available at Github [DM] to

[1] generated using David A. Wheeler's SLOCCount

allow other researchers to benefit from our experiences. With PAIN we assess both the achievable parallel experiment throughput as well as the validity of obtained results.

We continue in Section 3.2.2 by stating the research questions we are interested in, followed by the description of our overall experimental design in Sections 3.2.3 to 3.2.6 and the description of our experimental methodology in Sections 3.2.7 to 3.2.9. Section 3.3 reports our experimental results and provides an analysis of the observed effects, with Sections 3.3.3 to 3.3.5 summarizing our insights, discussing threats to the validity, and concluding our PAIN study.

3.2.2 Research Questions

Our goal is to assess the feasibility of using virtual machines for the parallel execution of SFI experiments on the lower levels of the software stack to increase test execution efficiency, but without adversely affecting experiment results. To that end, we investigate the following research questions.

RQ 1. *Can the throughput of SFI experiments be increased by execution them in separate VMs on the same host machine?*

RQ 2. *Can the execution of SFI experiments in parallel VMs on the same host machine change the obtained experiment results?*

RQ 3. *Assuming the answer to both RQ 1 and RQ 2 is yes, can the experiment setup be tuned for increased parallel throughput while avoiding result distortions?*

3.2.3 System Model

We investigate the impact of VM-based parallelism in the context of robustness assessments of OS kernels. We focus on device drivers in our experiments as they have been shown to contain more defects than other kernel code [Cho+01; Pal+11] as already mentioned in Chapter 2. Unfortunately, device driver failures have severe consequences on the overall system as they often crash the OS [GGP06; Sim03]. Injecting faults into drivers while observing kernel behavior helps to identify critical faults and gives useful feedback for robustness improvements of the kernel [Arl+02; NC01].

In our experiments, we automatically generate faulty device driver versions, load them into the kernel, and execute a workload to exercise the driver code. The target system is executed within a VM and all experiment control logic is run outside of this VM to ensure that experiment control cannot be corrupted by injected faults. Our PAIN framework distinguishes and detects the following failure modes:

- **SC**: *System Crashes* – detected by monitoring kernel messages

- **SE**: *Severe System Errors* – detected by monitoring kernel and VM messages
- **WF**: *Workload Failures* – detected by monitoring application logs
- **IHA**: *Init Hang Assumed* – detected if system boot-up and initialization takes longer than a timeout threshold
- **SHA**: *System Hang Assumed* – detected if workload execution did not start after a timeout threshold
- **WHA**: *Workload Hang Assumed* – detected if workload execution takes longer than given a timeout threshold
- **SHD**: *System Hang Detected* – detected by monitoring kernel internal metrics of the target system
- **WHD**: *Workload Hang Detected* – detected by monitoring kernel internal metrics of the target system

The SC and SE detectors are external to the VM that executes the target system. They read and analyze log messages emitted by the target system's kernel and the VM itself that executes the target system. An example for an SC failure is a kernel panic and for an SE failure a crash of the VM itself.

In addition to these two detectors, our setup also employs timeout based external detectors to detect hangs of the target system, i.e., periods without progress and no other failure indication. Our detectors assume such hangs in various stages of the experiment execution (IHA, SHA, WHA) if the execution of the respective stages takes longer than the provided timeout thresholds. The timeout thresholds have been calculated by measuring the execution time of the respective experiment stages without faults injected and then adding an ample safety margin to the measured values.

External detectors that rely on timeout values are known to have precision and efficiency issues as their threshold values can easily be set to inefficiently high or too small values, leading to wrong detections [Bov+11; CNR09; Zhu+12]. For that reason, we also make use of two additional, more advanced internal hang detectors similar to those from Zhu et al. [Zhu+12]. The two detectors execute within the target system, with a light detector running as user process that monitors system load statistics and a heavy detector that executes inside the target kernel. If the light detector senses a potential stall, it triggers the heavy detector for a more accurate assessment. The heavy detector then analyzes kernel internal metrics and, if an actual hang is detected, triggers a controlled system crash. The tests performed by both the light and heavy detectors are the same as those suggested by Zhu et al. [Zhu+12]. Note however, that the improved hang detectors cannot be used for the detection of hangs during system initialization because the internal detectors can be loaded only after the target system is fully initialized.

3.2.4 The SFI Fault Model

In SFI, the fault model specifies the introduced corruptions. We consider the injection of code mutations, i.e., changes in source code, in device drivers for emulating residual software defects in device drivers; this is similar to recent studies on software fault tolerance [NC01; SBL03] and on dependability benchmarking [DM03; DVM04; VM03]. For that purpose, we rely on the SAFE tool from Natella et al. [Nat+13], which is freely available [Nat13] for research purposes, to produce realistic code mutations that were derived from actual software defects found in commercial and open-source OSs [CC96; DM06].

As faulty drivers are notorious for seriously threatening system stability, the target system needs to be executed in strict isolation such that:

1) Experiments cannot affect the host system or the experiment control logic

2) Subsequent experiments can always start from a clean state that is free from any residual effects of previous experiments

These requirements results in high overheads for individual experiments and decrease achievable experiment throughput when executed sequentially, which is why parallelization is desirable to compensate for said overhead.

Moreover, parallelization should also mitigate the high volume of experiments that are needed for comprehensive assessments, especially when higher order faults, i.e., multiple faults at once, are employed. To emulate such a high volume scenario, we repeatedly applied the SAFE tool to driver code that already was mutated to produce higher order mutations as used in Higher-Order Mutation Testing approaches [JH09].

3.2.5 Measures for Performance and Result Accuracy

Performance Measure

We (and others [Ban+10; Han+10]) argue that a higher throughput of SFI experiments is worthwhile for achieving a higher coverage of fault conditions for testing. Hence, experiment throughput, i.e., the executed average number of experiments per hour, is the metric of interest.

Accuracy Measures

In contrast to the simple performance measure, we define the accuracy of SFI results in statistical terms because SFI experiments on the OS layer are heavily influenced by non-deterministic factors. For observing the effects of injected faults, the mutated code has to be activated, i.e., actually executed, during an experiment [GT07]. As hardware abstraction and mediation for hardware access are core functions of

the kernel, there is usually no direct interface to individual driver functions for programmers. Hence, a complex software layer interposes between device drivers and user mode applications. Many functions, such as power management, are hidden from user programs and activated by the OS upon commonly unpredictable hardware events and task scheduling decisions.

We measure result accuracy along two dimensions. First, we want to assess if the result distributions of failure modes change when we increase parallelism. For that purpose, a binary measure indicating statistically significant deviations is adequate. We rely on a χ^2-test for independence (with a significance level of $\alpha = 0.001$) to decide whether observed result distributions for parallelized experiments differ from the ones obtained from sequential experiment executions.

Second, we want to assess the stability and reproducibility of obtained results, which is why we measure result heterogeneity for repeated experiments at the same degree of parallelism as a comparative metric. We measure the variance of each observed result distribution by interpreting it as a vector in n-dimensional space and calculate the Euclidean distance from the mean of all observed distributions. We then compute the mean value of all such distances for all repetitions with the same configuration as the heterogeneity metric d.

3.2.6 Hypotheses

On this background, we formulate hypotheses derived from the research questions stated in the beginning of this section. We only state the null hypotheses to be tested; the alternative hypotheses are simply the negations of them.

Hypothesis H_0 1. *If the number of parallel experiments executing on the same host is increased, the experiment throughput does not increase.*

Hypothesis H_0 2. *If the number of parallel experiments executing on the same host is increased, the observed result distribution of failure modes is independent from that increase.*

Hypothesis H_0 3. *If the number of parallel experiments executing on the same host is increased, the heterogeneity among repeated experiments with the same configuration does not increase.*

3.2.7 Target System

We are conducting our experiments on the Android OS [Gooa], which is used in numerous different contexts, most prominently on smartphones. We use Android 4.4.2 "KitKat" with a Linux 3.4 kernel from the official Google repositories [Gooc]. We run the system inside the Goldfish System-on-Chip emulator [Goob], which is based on the QEMU emulation and virtualisation platform [Bel17] and ships

with the Android software development kit. We target the MMC driver, which consists of 435 source lines of code, for the emulated SD card reader of the Goldfish platform for our SFI experiments. We rely on a synthetic benchmark workload for exercising the MMC driver, which is based on code from Roy Longbottom's Android benchmarks [Lon].

Our workload reads and writes files on the SD card to exercise the MMC driver while generating additional CPU and memory load. We make use of code from the DriveSpeed, the LinpackJava, and the RandMem benchmarks and configure them to exercise the SD card driver for about 30 s. All three benchmarks run as parallel threads and we use additional threads in the benchmark apps to detect workload failures (WF), as application failures are signaled as Java exceptions and need to be explicitly forwarded to our external failure detectors.

3.2.8 Fault Load

We apply the SAFE tool repeatedly to the MMC driver source code to generate both first order and second order mutants. After the first SAFE application, we generated 273 mutants. We then generate further 70 167 second order mutants by applying SAFE a second time to each of the first order mutants. In total, this yields 70 440 faulty versions of the MMC driver.

For our experiments and analysis, we restrict ourselves to a subset of the generated mutants and randomly sample 400 mutants from the set of first and second order mutants. We repeat all our experiment campaigns three times for each experiment configuration to account for factors of non-determinism as we are using a complex OS level setup.

3.2.9 Execution Environment

We run our parallel experiments on the same host machine. According to the desired degree of parallelism, we replicate multiple instances of the Goldfish emulator, which executes the target Android system, on a single host machine. This approach to parallelization by replication reflects the assumption of non-interference between individual tests that we are questioning and is the same strategy employed in recent approaches to test parallelization [Ban+10; Han+10; Mah+12; Yu+09; Yu+10].

We execute all our experiments on two different host platforms to avoid biasing our results due to effects from a single platform:

- **Desktop**: A desktop machine running Ubuntu 13.10 with AMD quad-core CPU ($N = 4$), 8 GiB main memory, and 500 GB hard drive with 7200 RPM
- **Server**: A server machine running CentOS 6.5 with two Intel Xeon octa-core CPUs ($N = 16$), 64 GiB main memory, and 500 GB hard drive with 7200 RPM

In order to avoid result bias due to different CPU features and frequencies, we disabled hyper threading in the Intel CPUs[2], disabled power and performance optimizing features such as frequency scaling, and set all CPU cores to the same frequency of 1.8 GHz, which was the only common value that could be set on both hosts.

The degree of parallelism P_n, i.e., the number of experiments executing in parallel on the same host, was initially set to:

a) $P_n = 1$, i.e., sequential execution

b) $P_n = 2N$, with N being the total number of physical cores in the host

Note that, for brevity, we also write P_1 for $P_n = 1$ and accordingly P_{2N} for $P_n = 2N$. $2N$ is a common choice to maximize hardware utilization since using more instances than available physical cores increases the chances of each core to be utilized while some processes may be blocked, e.g., due to pending I/O operations. Increasing P_n further often leads to overload situations that may degrade the overall system performance drastically.

One workload, two host platforms, and two degrees of parallelism yield a total of 4 distinct experiment configurations. For each of these, we execute an experiment campaign of 400 experiments with three repetitions. Overall, we execute 12 campaigns with 400 experiments each to investigate our stated research questions. In total this sums up to 4800 individual experiments. We report the results of our experiments in Section 3.3, and augment the described setup with some additional experiments for further analysis.

3.3 PAIN Results and Analysis

In the following, we first present the results of our initial experiments and answer our first two research questions by rejecting or accepting our stated hypotheses. We then continue with further experiments to investigate our third research question by fine-tuning our experiments and reiterating over our hypotheses.

3.3.1 Initial Results

The results of our initial experiments are documented in Tables 3.1a and 3.1b on the next page as mean values over 3 repeated campaign runs. The *Setup* columns describe the used host platform and the used degree of parallelism P_n. The *Failure Modes* columns report the number of experiments that resulted in the respective failure modes. Note that in addition to the failure modes defined in Section 3.2.3, we report two additional modes as possible experiment outcomes: *Invalid* and *NF* (no

[2]The used AMD CPUs do not provide equivalent symmetric multithreading features (SMT).

Table 3.1: Results for our initial 12 experiment campaigns. The reported values are means over 3 repeated runs.

(a) Mean Failure Mode Distributions

Setup		Failure Modes									
Host	P_n	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA
Desktop	1	0.00	108.00	97.00	0.00	182.00	0.00	0.00	0.00	6.33	6.67
Server	1	0.00	114.67	97.00	0.00	183.00	5.33	0.00	0.00	0.00	0.00
Desktop	8	0.00	1.00	96.67	0.00	6.33	10.00	0.00	1.00	281.67	3.33
Server	32	0.00	65.00	97.00	0.00	179.00	5.00	0.00	0.00	48.00	6.00

(b) Performance and Accuracy Measures

Host	P_n	Throughput (exp./h)	Experiment Duration (s)	d
Desktop	1	12.5	286.97	2.02
Server	1	12.2	295.36	0.63
Desktop	8	56.1	493.77	3.50
Server	32	115.6	616.19	6.35

failure). The former is introduced to account for rare cases where the experiment control prematurely aborts experiments due to unexpected errors within the logic or issues on the host machine, such as memory shortage. The latter simply accounts for experiments that completed without any indication of failures, i.e., cases where the mutation within the MMC driver had no observable effect or was not activated. The column *Throughput* reports the average number of experiments that completed per hour. *Experiment Duration* reports how long a single experiment took and the d column reports the Euclidean distance measure of heterogeneity that we defined in Section 3.2.5.

Inspecting the achieved experiment throughput in Table 3.1b, we can see a clear increase in the average throughput when parallelism is increased from P_1 to P_{2N}. We therefore *reject* our Hypothesis 1. For the desktop host, we observe a speedup of about 4.5× when going from P_1 to P_8. For the server host, we see a speedup of about 9.5× when increasing parallelism from P_1 to P_{32}. It is interesting to note that, despite the overall speedup, the average duration of *individual* experiments increased, i.e., slowed down, for both hosts, by 1.7× for the desktop and by 2.1× for the server host.

To test Hypothesis 2, we perform a χ^2-test for independence to assess if the result distributions are statistically independent from the degree of employed parallelism, i.e., whether the distributions change when parallelism is increased

Table 3.2: χ^2-test of independence for parallelism (P_1 vs P_{2N}) and initial result distributions (cf. Table 3.1a).

Host	p	r	Verdict
Desktop	0.0	0.90	reject
Server	4.3×10^{-41}	0.40	reject

with the distribution for P_1 as our baseline. Note that for our χ^2-tests, we correct the obtained p-values according to the Benjamini-Hochberg procedure [BH95] to account for the risk of false discoveries when we perform multiple tests on the same population. We report the results of our χ^2-tests in Table 3.2. The p column reports the obtained p-value from the statistical test and the r column reports the normalized Pearson coefficients. r gives an indication of how "strong" the correlation between the observed failure mode distributions for P_1 compared to P_{2N} is. To conduct the χ^2-test, we used the absolute numbers from the distributions and not the mean values. As the obtained p-values are well below our chosen $\alpha = 0.001$, we *reject* our Hypothesis 2, i.e., parallelism and results are *not* independent.

Finally, we use our measure for heterogeneity d as defined in Section 3.2.5 to test Hypothesis 3. We therefore compute, within the each set of 3 repeated runs, the Euclidean distance of each original result distribution to the mean distribution (reported in Table 3.1a) and report the mean of these distances as d in Table 3.1b on the preceding page. Comparing d between P_1 and P_{2N} for both hosts, we see an increase in heterogeneity of 1.7× for the desktop and 10.1× for the server host. We therefore *reject* Hypothesis 3.

Considering these results, we answer both RQs 1 and 2 with *yes* and proceed in the following with our investigation of RQ 3.

3.3.2 The Influence of Timeout Thresholds

In our initial experiments, we observed that the number of observed Invalid, SC, SE, WHD, and SHA outcomes do not significantly differ across all configurations. However, for WHA, and IHA outcomes, we observe major differences as exemplified by the 44.5× increase in WHA for the desktop host when going from P_1 to P_8, which corresponds to more than 70 % of experiments in an individual campaign having this outcome. The affected failure modes are detected by detectors involving timeout threshold values that have to be set as part of the experimental setup. Given that we observed a slow down of individual experiments by a factor of about two, we suspect these detections for the P_{2N} case to be false positives due to too restrictive timeout thresholds. We therefore generously increase the used timeout thresholds by a factor of 3 for our WHA, SHA, and IHA detectors. The increased timeout thresholds imply unnecessary long waiting times in cases of

Table 3.3: Results for our repeated experiments with increased timeout thresholds. The reported values are means over 3 repeated runs.

(a) Mean Failure Mode Distributions

Setup		Failure Modes									
Host	P_n	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA
Desktop	8	0.00	104.00	97.00	0.00	181.67	5.00	0.00	0.67	11.33	0.33
Server	32	0.00	114.00	97.00	0.00	181.67	6.67	0.00	0.67	0.00	0.00

(b) Performance and Accuracy Measures

Host	P_n	Throughput (exp./h)	Experiment Duration (s)	d
Desktop	8	47.0	587.25	5.41
Server	32	118.1	619.48	1.99

actual hangs, which may decrease overall experiment throughput if actual hangs occur often enough. However, as we only observed few hang detections for P_1, we assume those to be rare.

We performed another 6 experiment campaigns with P_{2N}, 3 repetitions per host machine, with the adjusted thresholds and report the results in Tables 3.3a and 3.3b. The result distributions in Table 3.3a show closer similarity to those of the P_1 runs (cf. Table 3.1a). We performed additional χ^2-tests with the new results as documented in the upper two rows in Table 3.5 on page 64. Although we cannot reject the independence of the parallelism degree and the result distribution for the server host, we still can for the desktop host. Inspecting our heterogeneity measure d in Table 3.3b, we see that while d decreased 3.2× for the server, it actually increased 1.5× for the desktop. Along with increased result heterogeneity, we also observe a 1.2× decrease in experiment throughput for the desktop host. Overall, the server host shows no statistically significant correlation between P_n and result distributions with adjusted timeout thresholds, while the desktop still does. The differences in the failure distributions that lead to this indication are mainly due to SHD, WHA, and IHA counts. We suspect that the desktop host is already in an overload situation at P_{2N} which is indicated by the decrease in throughput.

As the server host showed improved result accuracy without throughput decrease at P_{2N} while the desktop did not, we want to investigate if the server host shows similar degradation as the desktop if load is increased. Hence, we performed 12 additional experiment campaigns on the server at P_{36}, P_{40}, P_{44}, and P_{48}, with 3 repetitions each. The results are reported in Tables 3.4a and 3.4b on the facing page. Note the relatively large increase in throughput is likely caused by optimizations in

Table 3.4: Results for our highly parallel ($P_n > 2N$) experiments on the server host. The reported values are means over 3 repeated runs.

(a) Mean Failure Mode Distributions

Setup		Failure Modes									
Host	P_n	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA
Server	36	0.00	113.67	97.00	0.00	181.67	7.00	0.00	0.33	0.33	0.00
Server	40	0.67	113.00	97.00	0.00	180.00	8.00	0.00	0.67	0.67	0.00
Server	44	0.00	112.00	97.00	0.00	180.33	6.67	0.00	1.33	2.33	0.33
Server	48	0.67	104.67	96.00	0.00	177.67	11.00	0.00	2.00	5.00	3.00

(b) Performance and Accuracy Measures

Host	P_n	Throughput (exp./h)	Experiment Duration (s)	d
Server	36	157.1	712.11	2.16
Server	40	154.1	834.14	1.98
Server	44	143.0	951.52	3.54
Server	48	102.5	1069.03	6.81

our experiment logic that were necessary to scale the experiments up to higher degrees of parallelism as we had to add logic to regularly clean the host of temporary files leaked by our controller and the emulation platform. We performed additional χ^2-tests and report their results in Table 3.5 on the next page in the lower four rows. The test results show that for parallelism degrees below P_{44} the result distributions are independent from the parallelism degree, but for P_{48} they are not. Moreover, we observe a large drop in throughput and an increase in heterogeneity for P_{48} (cf. Table 3.4b). Despite the results for P_{36}, P_{40} and P_{44} being more heterogeneous than for P_1, we still deem them acceptable compared to the increase in throughput.

3.3.3 Discussion

In the following, we summarize and discuss the main lessons learned from our experience with conduction parallel FI experiments. A surprisingly important aspect of our PAIN experience was:

> *It can be difficult to correctly setup and fine-tune parallel FI experiments and this requires special care.*

Table 3.5: χ^2-test of independence for parallelism (P_1 vs P_n) and additional result distributions.

Host	P_n	p	r	Verdict
Desktop	8	6.7×10^{-7}	0.18	reject
Server	32	1.0	0.05	do not reject
Server	36	1.0	0.05	do not reject
Server	40	0.78	0.08	do not reject
Server	44	0.21	0.10	do not reject
Server	48	1.3×10^{-4}	0.17	reject

The parallelization of the experiment execution had a significant impact on the timing behavior and duration of individual experiments. Moreover, we observed incorrect false positive detections of our timeout-based detectors in our initial experiments. Although we did our best to find sensible timeout thresholds by performing initial fault-free parallel runs, these fault-free runs could not account for unexpected delays and interactions among VMs when executing faulty driver versions. Furthermore, we had to invest considerable effort into the testing and debugging of our PAIN framework that by itself is a complex and concurrent piece of software.

The framework related issues, we had to deal with, included: resource leaks (e.g., temporary files) of the used Android emulator, portability problems across the server and desktop host, for instance, the server host had lower *rusage* limits on processes and memory consumption that we could debug only after extensive inspection of kernel log facilities, and synchronization and communication issues between experiment control logic and emulator as the timing behavior of the emulator was unreliable and sometimes messages were lost due to ordering issues. Based on our own experience with these complexities, we advise others designing similar parallel setups to pay very close attention to these aspects.

Revisiting the research questions we started from, our experiments indicate that, while parallelism can improve throughput, it can also affect the obtained results. Running experiments in parallel can significantly (RQ1) increase experiment throughput, in our case we observed speedups up to 10× for the server host with P_{32} (tripled timeouts). Hence, FI experiments can benefit from the available parallel hardware resources. However, we also observed that:

> *The parallel execution of FI experiments can significantly improve throughput, but at the same time adversely affect result accuracy.*

Our analysis shows that there can be statistically significant differences between the result distributions obtained from sequential compared to parallel experiment execution (RQ2), i.e., using parallelism changes experiment outcomes. The same is true for result stability among multiple repetitions as the heterogeneity potentially increases with parallelism. Hence, there is a risk of such effects influencing the conclusions drawn about dependability properties of a target system from such experiments. In our experiments, performance interferences between concurrently executing VMs changed the observed failure modes of some of the experiments to hang failures, which were not easily reproducible across repetitions and led to unstable result distributions. We observed these issues for higher degrees of parallelism, for instance, for P_{3N} (three times the number of cores) for our server host. However, using a lower degree of parallelism for experiments does not lead to statistically significant deviations for parallel executions compared to sequential ones, for instance, on the server, we could run experiments at P_{32} without such problems once the timeout thresholds were adjusted. This suggests that parallelism does not in general harm result accuracy (RQ3) given that the parallelism degree is sensibly chosen.

For maximizing experiment throughput and at the same time preserving the accuracy of obtained results, they employed degree of parallelism has to be carefully chosen. For instance, we achieved the best throughput on our server host at P_{36} with 157.1 experiments per hour. But with increasing degrees of parallelism, the throughput decreased down to 102.5 at P_{48} and results became unstable and inaccurate. This exemplifies that results start deviating once a system becomes overloaded with too many parallel experiment instances:

> *If the degree of parallelism is carefully chosen to achieve the best experiment throughput, adverse effects on the accuracy of the results can be averted.*

Therefore, it must be considered an important task of the tester to ascertain a suitable degree of parallelism for conducting parallel experiments. For instance, it is a viable approach to conduct a number of preliminary experiment runs with increasing degree of parallelism to determine at which point the system shows indications of overload and therefore degraded experiment throughput. Although our study was focused on FI experiments using VMs, the discussed issues also occur in other forms of FI and testing in general, for instance, if testing frameworks rely on fixed timeouts per test, and other complex systems involving unpredictable or non-deterministic timing patterns, as often the case in concurrent software.

3.3.4 Threats to Validity

As with any empirical study, we must be careful when interpreting the obtained results and drawing conclusions. We identified the following main threats to validity: the selected injection target, the employed fault model, the used workload, and the used measures for result accuracy.

As injection target, we used the Linux kernel of an Android OS. This system may not be representative for all kinds of conceivable software systems, however, it is a good representative for embedded, real-time and systems software, being important targets for FI. In addition, being a real-world complex OS, this setup involves many factors of non-determinism whose influence we were interested in investigating. Examples of such factors include: I/O interactions, external hardware events, concurrency, and non-determinism due to scheduling and memory management.

We use a fault model that mutates software code to inject representative software defects as supported by extensive analyses and being widely accepted in practice [ABL05; DM06; DR06; Nat+13]. We generated first and second order mutants for having a large sampling base for our parallel experiments. Although the representativeness of repeated mutations has not yet been investigated in detail, using multiple fault injections is already used in practice [JGS11; JH09; Win+13].

We rely on existing performance benchmarks for the Android OS as workload. Even though benchmarks may not necessarily be representative of user scenarios, they are widely used in FI studies, in particular for dependability benchmarking [KS08; NC01; VM03]. Moreover, previous studies have shown that stressful workloads like performance benchmarks increase the likelihood of activating faults [Tsa+99].

For our experiments and analysis, we rely on accuracy measures that are focused on the observed distributions of failure modes. Said distributions are a highly important aspect when conducting SFI experiments that often want to assess the likelihood of failure modes with high severity [Avi+04; DM03; KD00; NC01]. Moreover, measurements of fault-tolerance properties, e.g., coverage or latency, depend on the concrete failure types that are observed during experiments. Consequently, it is important for dependability assessments to avoid distortions of the observed failure mode distributions.

3.3.5 Concluding Remarks

With the increasing complexity of software, FI experiments become more complex and require more injection experiments [Gun+11; JGS11; Lan+14; Win+13]. In the previous sections of this chapter, we investigated if the parallel execution of such experiments is a feasible strategy to cope with increasing number of experiments, relying on our PAIN framework for PArallel fault INjections. Our PAIN experiments on the Android OS showed that parallel execution in separate

VMs can significantly increase experiment throughput, but also adversely affect the obtained results. We found that the degree of employed parallelism as well as timeout thresholds for timeout-based failure detectors must be carefully chosen to prevent resource contention and the timing of events from distorting result distributions. If the machine that hosts the experiments gets overloaded, results become inaccurate.

3.4 FastFI Approach

After we concluded our PAIN study in the previous sections, we now continue with the presentation of our design an implementation of FastFI that is heavily inspired by insights we gained from PAIN. In contrast to the PAIN framework, which relies on VMs, FastFI employs processes to execute experiments on higher levels of the software stack. FastFI combines multiple techniques to reduce the overall execution latencies of SFI tests by:

1) not re-executing redundant code paths,

2) reducing the number of tests without fault activation, and

3) parallel execution of tests.

Section 3.4.1 gives an overview of FastFI and its workflow while the following Section 3.4.2 details the FastFI execution model. Section 3.4.3 provides a detailed discussion of the employed parallelization strategy and the required control logic. Sections 3.4.4 and 3.4.5 discuss the required static analysis and technical limitations of the approach. Section 3.4.6 provides a brief overview of our prototype implementation. We then continue to our evaluation in Section 3.5.

3.4.1 Overview

For SFI tests, faulty versions of a given software are generated, which are then executed separately and the outcome of their execution is monitored. These test executions typically require external experiment logic for controlling which faulty versions get executed and for monitoring test outcomes. Typical test outcomes include successful execution, execution with error indication, and aborted (crashed) execution. Note that each faulty version typically contains only one single fault to allow for the isolated observation of the fault's effects. For tests with fault combinations (higher order faults), additional software versions need to be generated, leading to a combinatorial explosion of the number of separate software versions.

With FastFI, the generated faulty versions of the software are not built and executed separately, but they are integrated into one test executable. For that

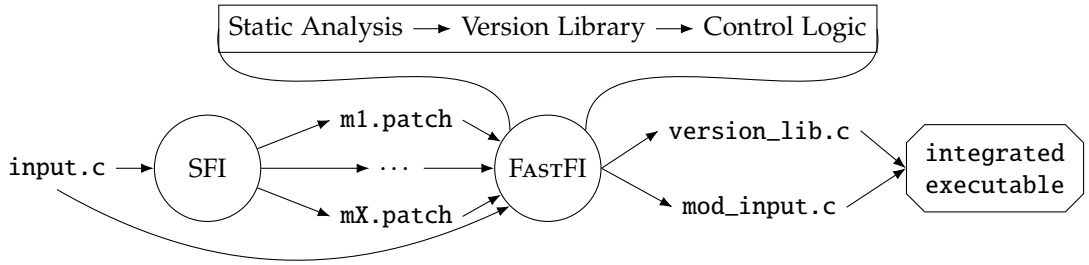


Figure 3.1: Overview of the FastFI workflow. The input is the original source code and SFI mutation patches. The final output is a FastFI-enabled integrated executable.

purpose, FastFI groups the faulty program versions by the functions in which the faults are injected. Each fault is then included in the program as a faulty version of the function that it modifies, rather than creating faulty versions of the complete program as in existing SFI tools. Although the fault grouping granularity could be changed from the function level to, e.g., basic block or even statement level, function-based grouping appears to be the natural choice for procedural languages and proves effective in our evaluation (see Section 3.5). The FastFI runtime controls *on demand* which of the integrated faulty versions are executed once the test execution reaches a point where a faulty function version can be selected for execution. The executions of the different faulty versions are isolated from each other by forking a new process for each faulty version. The FastFI runtime includes all control and monitor logic needed to conduct SFI tests, i.e., no external logic is required to conduct a full set of tests with all generated faulty software versions.

Figure 3.1 provides an overview of the FastFI tool chain that generates a FastFI-enabled test executable for a given software. The FastFI tool chain takes the original source code and the code mutation patches from an SFI tool as input. Similarly to our PAIN setup, we use SAFE [Nat+13] as SFI tool in our evaluation, but FastFI is independent of the actual SFI tool used. The only constraints are that the code patches generated by the SFI tool modify only one source code function at a time, as FastFI groups faults on a per-function level, and that the patch files adhere to the commonly used (unified) diff format as understood by the GNU `patch`[3] tool. The FastFI tool then performs the following steps on the provided inputs:

1. Static source code analysis for function extraction and fault grouping
2. Generation of a code library with all faulty function versions
3. Insertion of the FastFI fork server control logic into the original functions

[3] http://savannah.gnu.org/projects/patch

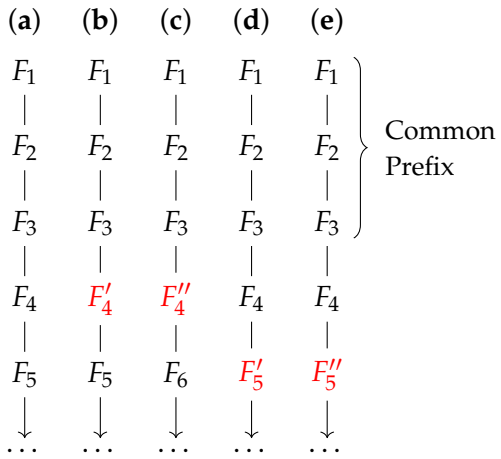


Figure 3.2: Traditional Execution Model. F_i denote functions and F'_i denote faulty versions of a function.

The output is a modified version of the original source code with the FastFI fork server control logic inserted and a library of all faulty function versions as well as copies of the original, unmodified functions. The final output after the usual software build process is the integrated FastFI-enabled test executable.

3.4.2 FastFI Execution Model

FastFI introduces a novel, more efficient execution model for SFI tests that is enabled by the integration of all faulty software versions into one integrated executable. Both the traditional and the novel FastFI model are discussed and contrasted in the following.

Traditional Execution

In the traditional execution model for SFI tests, each faulty software version is an executable of its own that has to be compiled and executed separately. Figure 3.2 illustrates an example for the execution of 5 tests in the form of function-level execution traces. The F_i are the functions executed. Faulty versions are marked with prime symbols, e.g., F'_4 denotes a faulty version of function 4 and F''_4 denotes another faulty version of the same function. Trace (a) represents the execution of the original, fault-free software whereas traces (b) and (c) represent executions with faulty versions of F_4 and traces (d) and (e) with faulty versions of F_5. Each execution trace can contain only one faulty version of any function. However, the same faulty version of a function can obviously be invoked more than once during an execution. All the different traces share a common execution prefix up to the point where a faulty function is invoked for the first time. In the illustrated example, F_1 to F_3

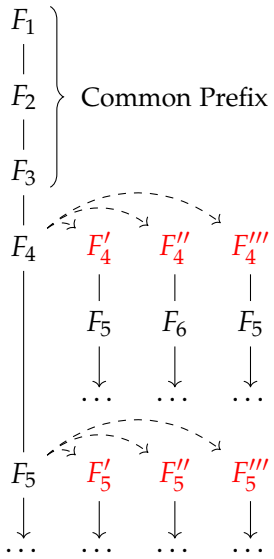


Figure 3.3: FastFI Execution Model. F_i denote functions and F'_i denote faulty versions of a function. Dashed arrows represent process forks.

is the common execution prefix for all 5 traces. For traces (a), (d), and (e), the common prefix is F_1 to F_4 as the first invocation of a faulty function happens later in the execution. After the invocation of a faulty function, the different executions may deviate drastically depending on the injected fault type and on whether it is activated during execution of the faulty function, as faults may arbitrarily change the program state. For instance in trace (c), F_6 is invoked after the fault in F''_4 was activated instead of F_5 as in the fault-free execution (a) or after execution of the faulty function F'_4 in (b). Hence, although there is a common execution prefix between tests, there is generally no common postfix once a fault has been activated. However, re-executing the common prefix for each individual test is time-consuming redundant work that can be avoided using FastFI as detailed in the following section.

FastFI Execution

The essential difference to the traditional execution model is that FastFI does not re-execute the common execution prefixes for all faulty versions and selects the faulty versions to be executed *on demand* during runtime. The FastFI execution model is enabled by the integration of all faulty versions into one executable. Figure 3.3 illustrates an example for the execution of 7 tests as function-level execution traces similar to the illustration for the traditional model in Figure 3.2. The common prefix F_1 to F_3 is only executed once in this model by the master process, which is represented by the leftmost execution trace. The master process

controls the execution of the faulty versions but never executes a faulty version of any function itself. Instead the master process creates, i.e., forks, (illustrated by dashed arrows) new child processes that execute the faulty versions on its behalf. Each faulty version is executed in its own process. In the example, once the master execution reaches F_4, for which three faulty versions exist, the FastFI runtime forks a new process before the faulty version F_4' is invoked. Since the `fork` system call creates an exact copy of the calling process, the new process starts its execution right were the master process called `fork` and invokes F_4'. Because F_4' now executes in its own process, it cannot interfere with the execution of the master process. Both processes are isolated from each other by means of operating system process isolation, which, for instance, guarantees memory isolation such that the fault executing child process cannot write to the master process' memory. However, there is still the possibility of interference via external resources that are not covered by OS process isolation such as the shared file system. Therefore, the FastFI runtime does not only perform a `fork` but also takes actions to minimize the chance for interferences via open files by file descriptor manipulation and I/O redirection upon forking.

Once the child process that executes the faulty version has finished, the master process either continues with the execution of the next faulty version, if available, or proceeds with its own fault-free execution. In the example, the master continues with the execution of F_4'' and afterwards F_4''' in their own processes and then proceeds in its own execution by invoking the original F_4. When the master process eventually finishes, the master's execution corresponds to the execution of a fault-free software version and all faulty versions that were reachable have been executed. Since software functions that are not reachable during execution, for instance, if the provided program input does not trigger all of them, are never executed by the master process, they do not impose additional test latencies. This is an improvement over the traditional execution model, because it is generally not known a priori which faulty versions are reachable during execution. Hence, FastFI automatically reduces the amount of faulty software versions to execute to the amount that is reachable and, thereby, avoids the execution of superfluous tests.

In addition to the test latency reduction due to the efficient execution of common prefixes and the automatic reduction of the number of faulty versions that need to be executed, FastFI reduces latencies further by allowing for the parallel execution of faulty versions of the same function. In the example in Figure 3.3, all faulty versions of F_4 can be executed in parallel. The same is true for F_5. As faulty versions need to be executed in their own processes in any case, there is no additional cost associated in executing them in parallel. The following section details the parallelization strategy employed as well as the control logic required to implement the FastFI execution model and the monitoring of the child processes executing faulty versions.

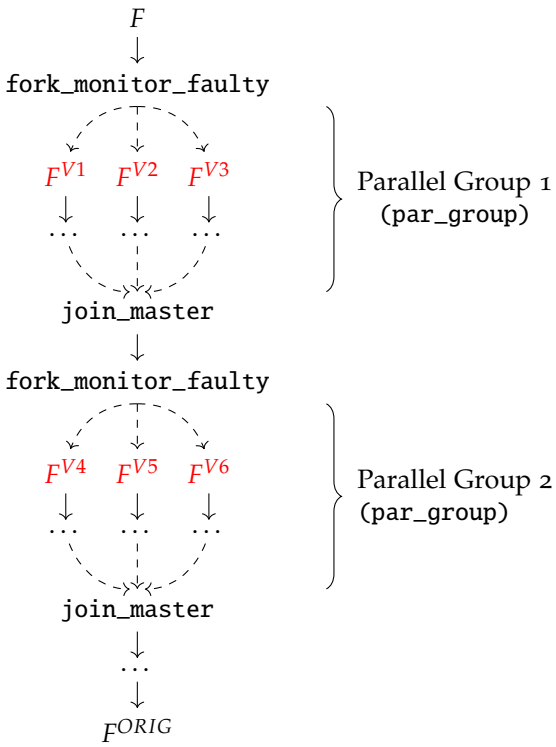


Figure 3.4: FastFI Parallel Execution. The example illustrates the execution of all versions of function F using parallelism degree $P_n = 3$ from the perspective of the master process.

3.4.3 FastFI Fork Server: Control & Monitoring of Faulty Versions

We denote the control logic that is responsible for implementing the FastFI execution model, which we described in the previous section, as the FastFI fork server. The fork server replaces the function body of all functions for which faulty versions exist. For each such function, distinctive fork server code is generated that controls which version gets executed and which degree of parallelism is employed.

Parallelization Strategy

FastFI parallelizes the execution of faulty versions by grouping all versions of each function into groups of size P_n where P_n denotes the degree of parallelism used, i.e., the number of faulty versions that may execute in parallel. P_n is a runtime parameter that can be chosen by the user for each run of the integrated FastFI executable. Each of the parallel groups is executed concurrently by forking all P_n versions at once. Before executing the next group, the previous group has to finish. Figure 3.4 illustrates an example for the execution of some function F, which is executed with parallelism degree $P_n = 3$. Once function F is invoked by the master

process for the first time, all its versions need to be executed before the master process can eventually execute the fault-free version F^{ORIG}. Hence, the master executes all parallel groups for F sequentially until all groups have been executed. However, the members of each group are executed in parallel.

The faulty version groups are generated by dividing the list of all versions into consecutive non-overlapping chunks of size P_n. The last group generated may be smaller than P_n if the number of versions is not evenly divisible by P_n. The total execution time of all faulty versions of a function is determined by the longest execution times among the versions executed within each parallel group. The total execution time is minimized if all members of a group have similar execution times. Therefore, the list of all versions should ideally be ordered according to expected execution times before chunking. Since this information is generally not know ahead of time, i.e., before actual execution, we order the version list according to the mutation operators that were applied to generate the faulty versions. In our experience, faulty versions that have been generated by the same mutation operators often show a tendency to result in similar execution times.

Control Logic

The FastFI fork server control logic replaces the function body of each function for which faulty versions exist. The original function versions are saved in the version library for each function and can still be invoked by both the master process as well as fault executing processes.

The listing in Figure 3.5 provides a simplified description of the FastFI fork server logic for some function `foo` in the form of C-like pseudo code. The fork server logic starts in lines 2 to 5 by verifying if the FastFI execution model should be used or if the user requested a traditional execution in which only one version, which is chosen by the user, gets executed. Both the execution mode and the requested version to execute are runtime parameters that can be configured by the user upon each execution of the integrated FastFI executable. This feature allows testers to investigate the behavior of individual faulty versions in detail without the overhead of re-compilation. In the traditional execution mode (discussed in Section 3.4.2), there is no distinction between a master and a fault executing process and no integrated monitoring is in place. In order to actually invoke a requested function version, the version is looked up in the version library and called dynamically as shown in lines 3 and 4.

The FastFI execution model is implemented by the logic in lines 6 to 22. The logic has to distinguish between three execution states as the master and all forked processes share the same code:

(1) in fault executing process (lines 6 to 10),

```
1  ret_type foo(args) {
2    if (in_single_version_mode) {
3      return call_version(
4          foo, args, requested_version);
5    }
6    if (forked) { // in faulty execution (1)
7      if (is_active(foo))
8       return call_version(foo, args, CUR_ACT);
9      else
10       return call_version(foo, args, ORIG);
11   } else if (!forked && already_done(foo)) {
12     // master: all versions done (2)
13     return call_version(foo, args, ORIG);
14   } else {
15     // master: exec faulty versions (3)
16     for (par_group in parallel_groups(foo)) {
17       fork_monitor_faulty(par_group);
18       join_master(par_group);
19     }
20     set_already_done(foo);
21     return call_version(foo, args, ORIG);
22   }
23 }
24
25 ret_type fork_monitor_faulty(par_group) {
26   for (cur_version in par_group) {
27     if (fork() == MONITOR) {
28       if (fork() == MUTANT) {
29         // run faulty version
30         forked = true;
31         CUR_ACT = cur_version;
32         setup_env();
33         return call_version(
34             foo, args, CUR_ACT);
35       } else {
36         // monitor faulty version
37         results = observe_wait(cur_version);
38         log(results);
39         exit_monitor();
40       }
41     }
42   }
43 }
```

Figure 3.5: Pseudo Code of the FastFI Fork Server Control and Monitor Logic.

(2) in master process after all faulty function versions have been executed (lines 11 to 13), and

(3) in master process upon the first function invocation (lines 14 to 22).

In state (1), the invocation of the correct version within a fault executing process is implemented; in state (2), the invocation of only the original, fault-free versions is guaranteed for the master process; in state (3), the actual selection and forking of faulty versions takes place.

In state (1), the logic has to distinguish whether a faulty version of the function (`foo` in the example) is active in the current process (line 7). If so, the correct faulty version from the library is invoked; if not, the original version is called. This guarantees that each fault executing process executes only one faulty version of any function.

State (3) corresponds to the situation exemplified in Figure 3.4 and discussed in Section 3.4.3, i.e., the actual forking of the parallel version groups happens here. The master process iterates over all version groups `par_group` (lines 16 to 19) and invokes the helper function `fork_monitor_faulty` for each of them, resulting in the execution of the faulty versions. Each loop iteration waits at the end until the execution of the current group finishes before starting the next iteration. After all faulty versions have been executed, the master process marks the function as done (line 20) to prevent redundant re-executions. As last step, the original function version is invoked (line 21) which finishes the fork server execution and advances the fault-free execution of the master process.

The actual forking logic is implemented in `fork_monitor_faulty` (lines 25 to 43). The function iterates over all P_n members `cur_version` of the current version group `par_group`. For each version `cur_version`, *two* processes, `MONITOR` and `MUTANT`, are created via `fork` calls. The `MONITOR` process, which has not been discussed so far, is required to perform reliable monitoring of the fault executing process. This monitoring needs to occur in a separate process since the fault executing process itself may behave erratically and, for instance, crash or hang indefinitely. The `MONITOR` process is created first (line 27) such that the `MUTANT` process becomes its child (line 28). Therefore, `MONITOR` can exercise process control over `MUTANT`. For instance, it can terminate `MUTANT` and it can observe crashes and exits of `MUTANT`. The `MONITOR` logic is shown in lines 36 to 39. `MONITOR` waits until the `MUTANT` process, which executes `cur_version`, finishes execution, or terminates it if execution takes longer than a user-specified timeout to ensure progress, fetches observed results and logs them for later analysis.

The `MUTANT` control logic is shown in lines 29 to 34. First, `MUTANT` marks itself as fault executing process (line 30) and remembers which version it is supposed to execute (line 31). Next, it performs additional environment setup steps (line 32) such as I/O redirection. As last step, `MUTANT` finally invokes the faulty function version

for the first time (line 33). At this point, `MUTANT` continues with the independent execution using the faulty version `CUR_ACT` upon each function invocation (`foo` in this example).

3.4.4 Static Analysis & Version Library Generation

FastFI requires knowledge about the static structure of both the input source code as well as the SFI mutation patches in order to be able to correctly replace function bodies, generate the FastFI fork server code, and to generate the library of faulty versions. To that end, FastFI relies on an existing static analysis framework to extract the necessary information about all functions present in the input source code. FastFI requires information about where functions reside in the source code, their function signature, and function parameter names. In a static analysis step, FastFI builds an analysis database with the required information for later use in the workflow as described in Section 3.4.1.

The mutation patches are parsed and information about modified source code lines are extracted. This information is then used to search the analysis database to match mutation patches to the functions that they mutate, i.e., the faults are grouped according to the source code function where they will reside. Once the grouping is complete, FastFI generates the library of faulty source code functions. For that purpose, each mutation patch is applied to the source code and the resulting faulty function is extracted, given a unique name, and added to the library. After each patch application, the original source version is restored to produce faulty versions that contain exactly one fault. As a final step, an unmodified version of each function is added to the library as well.

3.4.5 Limitations

We discuss technical limitations that may impede the application of FastFI in the following. Since FastFI relies on the `fork` system call as specified by POSIX, FastFI can be used only in environments where `fork` or a compatible system call is available. Moreover, an invocation of `fork` must leave both the calling and the created child process in a well defined state from which independent executions of parent and child processes are possible. This is not the case for multi-threaded processes as only the calling thread survives a `fork` invocation and the created process has only limited abilities to invoke further system services.

Software may behave differently under the FastFI execution model under certain circumstances. If the software's behavior depends on explicit process attributes, such as the process identifier (PID), its behavior may change as FastFI creates new processes with possibly changed attributes (e.g., different PIDs). Software that relies on explicit time information, e.g., by using timers or explicit time duration, may behave differently as FastFI effectively pauses the execution of the master

process while faulty versions are executed. Moreover, software that contains severe defects such as invalid memory accesses in the original program may have different effects in FastFI as the memory layout between the generated executables differs.

FastFI isolates the execution of faulty software versions by means of OS process isolation. This leaves external resources that are not covered by process isolation as possible sources of interferences. While FastFI handles open files, additional measures need to be taken to also handle hardware devices or network connections.

3.4.6 Implementation

We developed a prototype of FastFI for software that is written in the C language and executes in a POSIX compliant environment. Our prototype relies on Coccinelle [INR18; Pad+08] as static analysis framework for C source code. It is mainly developed in Python and can, as the evaluation in Section 3.5 demonstrates, efficiently handle real world software despite the fact that it is not yet optimized for performance.

Please note that, although our prototype currently only supports software written in C, FastFI itself is not limited to C software. Software written in other languages, such as C++ or Rust, can also benefit from FastFI.

3.5 FastFI Evaluation

In order to evaluate the applicability and performance of FastFI for real world software, we investigate the following research questions using our prototype implementation for C software.

RQ 1 How much can FastFI reduce overall test execution latencies for sequential SFI tests?

RQ 2 How does the execution speedup achieved by FastFI develop with increasing degree of parallelism?

RQ 3 Do SFI test results remain stable across runs with increasing degree of parallelism when using FastFI?

RQ 4 How large is the build time overhead of integrated FastFI builds compared to traditional separate builds?

3.5.1 Experimental Setup

Execution Environment

We conduct our experiments on a machine with up to date Debian Buster (Linux 4.16, x86_64) as operating system. The machine is equipped with an AMD Ryzen 7 CPU

Table 3.6: Overview of the PARSEC applications used in the evaluation.

Application	Description	Mutants
`blackscholes`	Numerical financial computations	416
`dedup`	Data stream compression	662
`ferret`	Content-based image similarity search	6157
`x264`	Video stream encoding and compression	13368

with 8 physical and 16 logical cores running at 3.40 GHz, 32 GiB of main memory, and a 1 TiB SSD.

Evaluation Targets

We apply FastFI to four applications from the widely used PARSEC benchmark suite 3.0 provided by Princeton University [Bie11; Pri09]. Table 3.6 gives a brief overview of the selected applications. We selected these four applications since they are representative for different application domains and they are written in C, which our current prototype implementation targets. We use the "simmedium" workloads that ship with PARSEC to exercise the applications. These workloads are of a moderate size, which allows us to execute our experiments within a reasonable time frame (within days).

Execution Steps

To investigate our research questions, we take the following steps for all selected evaluation targets.

We first apply the SAFE software fault injection tool [Nat+13; Nat13] to generate mutation patches. SAFE applies 13 different mutation operators to generate representative software faults. An overview of the generated mutants is given in Table 3.6. Each mutant creates a faulty software version that needs to be executed for SFI tests.

Next, we perform the static analysis of the input source code using Coccinelle to generate the analysis database as described in Section 3.4.4. We then analyze the generated mutation patches and perform the function level fault grouping. Afterwards, we generate the library of faulty versions by applying the mutation patches and extracting the resulting modified functions as well as saving the original, unmodified function version. Then, the original function bodies are replaced with the generated FastFI fork server code as described in Section 3.4.3. As final step, we build the integrated executable with the PARSEC default build configuration "gcc-serial" that results in non-multithreaded executables.

We perform our experiments using the generated integrated executables in our execution environment. We repeat each experiment 3 times and report averages.

3.5.2 RQ 1: Sequential Speedup

To determine the impact of FastFI on sequential SFI execution latency, we compare the performance of FastFI without any parallelization ($P_n = 1$) to the performance achieved by separately executing each faulty version. For the separate executions baseline, we make use of our single version mode as described in Section 3.4.3, i.e., we still use the integrated executables generated by FastFI. However, the faulty version to execute is picked prior to execution, and only one faulty version is chosen for each program execution. Consequently, executions in this mode do not benefit from the ability of FastFI to avoid redundant code execution and the execution flow corresponds to a traditional SFI execution model as described in Section 3.4.2.

As shown in the leftmost column of Figure 3.6, FastFI can achieve speedup factors from 1.3 to 3.6, depending on the benchmark. In the absence of parallelization, these speedups are the result of avoiding redundant code execution. FastFI avoids redundant code execution in two ways: (1) By efficiently executing common prefixes and (2) by automatically reducing the number of faulty versions that need to be executed. The reduction in the number of faulty versions is shown in Figure 3.7. For three out of four benchmarks, FastFI automatically executes fewer faulty versions than the traditional execution model as unreachable faulty versions are not executed. The maximum reduction can be observed for `ferret` where FastFI reduces the number of faulty versions down to 47.9 %. This substantial reduction is also reflected in `ferret`'s speedup factor of 3.6. Moreover, despite executing the same number of faulty versions, FastFI achieves a speedup of 1.3 over the traditional execution model for the `blackscholes` benchmark. This reduction is the effect of FastFI's efficient common prefix execution.

> FastFI*'s ability to avoid the execution of both "dead" faulty versions and redundant path prefixes significantly speeds up sequential test execution. We achieved a best case speedup of 3.6×.*

3.5.3 RQ 2: Parallel Speedup

To investigate how the speedup achieved by FastFI develops with increasing degrees of execution parallelism, we configure FastFI to run up to 32 faulty versions in parallel. Note that changing the degree of parallelism is handled by the FastFI runtime code and does not require recompilation (see Section 3.4.3). The

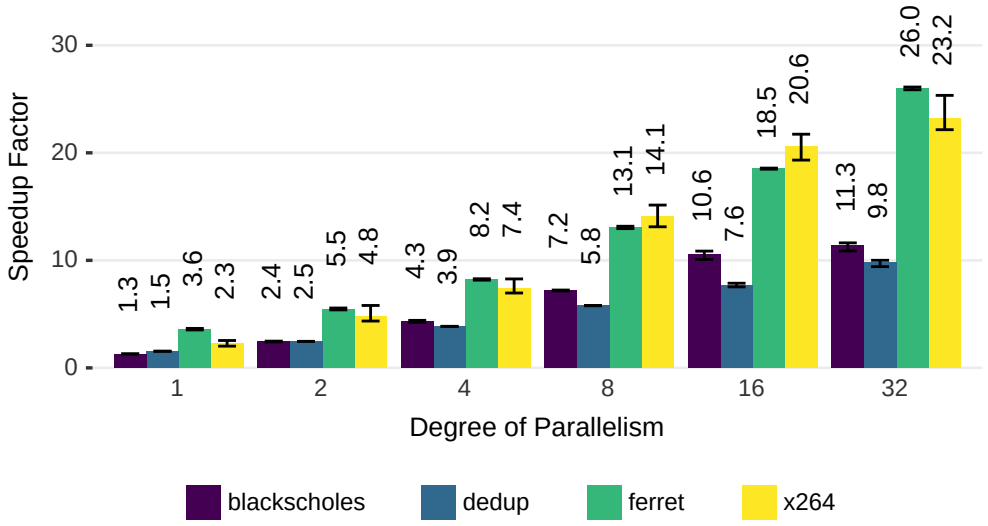


Figure 3.6: FastFI speedup relative to traditional execution model for increasing degrees of parallelization (P_n). Error bars indicate minimum and maximum speedup.

speedups relative to traditional execution for the different degrees of execution parallelism are shown in Figure 3.6. FastFI achieves increasing speedups with an increasing degree of parallelism. When executing 16 faulty versions in parallel, which corresponds to the number of logical cores on the machine we use for our evaluation, FastFI achieves a speedup of 7.6 to 20.6 compared to the traditional execution model. Relative to FastFI execution without parallelism, the speedups range from 5.0 to 8.9. When going beyond the number of available cores by executing 32 faulty versions in parallel, FastFI achieves speedups ranging from 9.8 to 26.0 relative to the traditional execution model, or 6.5 to 10.0 over FastFI execution without parallelism. These results show that parallel FastFI execution enables significant speedups over traditional SFI execution as well as over FastFI execution without parallelization. By optimizing the FastFI fork server architecture to allow for dynamic parallel groups (see Section 3.4.3), we believe that even higher speedups can be achieved.

> FastFI's *ability to execute multiple faulty versions at the same time in parallel processes significantly speeds up parallel test execution. We achieved a best case speedup of 26× for 32 parallel instances.*

3.5.4 RQ 3: SFI Result Stability

To determine whether increasing degrees of parallelism affect SFI result stability, we configure FastFI to run up to 32 faulty versions in parallel and compare

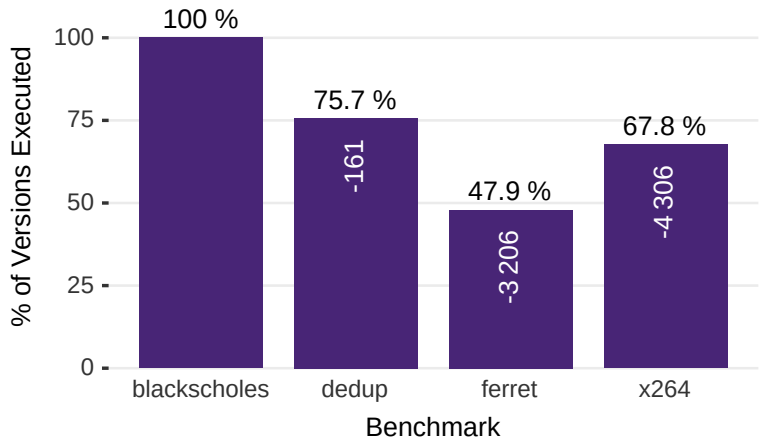


Figure 3.7: Percentage of faulty versions executed during (sequential) FastFI execution. The reduction is due to FastFI's ability to avoid execution of unreachable versions.

SFI test outcomes. We consider the common four classes of SFI test outcomes: "Crash": application crash, "Error": termination with error indication, "Success": termination without error indication, and "Timeout": application did not finish in time. From an application perspective, these failure modes match the crash and hang oracles that are most commonly applied for SFI and robustness tests [KDD08]. We set the timeout values to 3 times the duration of a fault-free execution for each benchmark to account for increased individual execution latencies in parallel testing that we observed in ourPAIN experiments.

Figure 3.8 shows the SFI test outcomes for different degrees of parallelism. The rightmost columns labeled with "s" show results from the sequential single version execution mode that corresponds to a traditional execution. The higher count of successful tests for this mode is due to the fact that all faulty versions are executed independent of whether the faults are reachable. Such "dead" versions always result in success as their execution always corresponds to a fault-free execution. Since FastFI avoids the execution of such "dead" versions, the success count for FastFI runs is lower.

For all benchmarks, the results are stable for up to 16 parallel executions. When executing 32 faulty versions in parallel, results remain stable for the `blackscholes` benchmark. For the other three benchmarks, the number of crashes and errors remain stable but the number of successful tests drops and the number of timeouts increases compared to lower degrees of parallelism. Moreover, for the `x264` benchmark, the number of successful executions and timeouts varies between test runs at this degree of parallelism. As this effect only occurs when running at a degree of parallelism well in excess of the available computational resources on the machine we use for our experiments, we expect that spurious timeouts at this

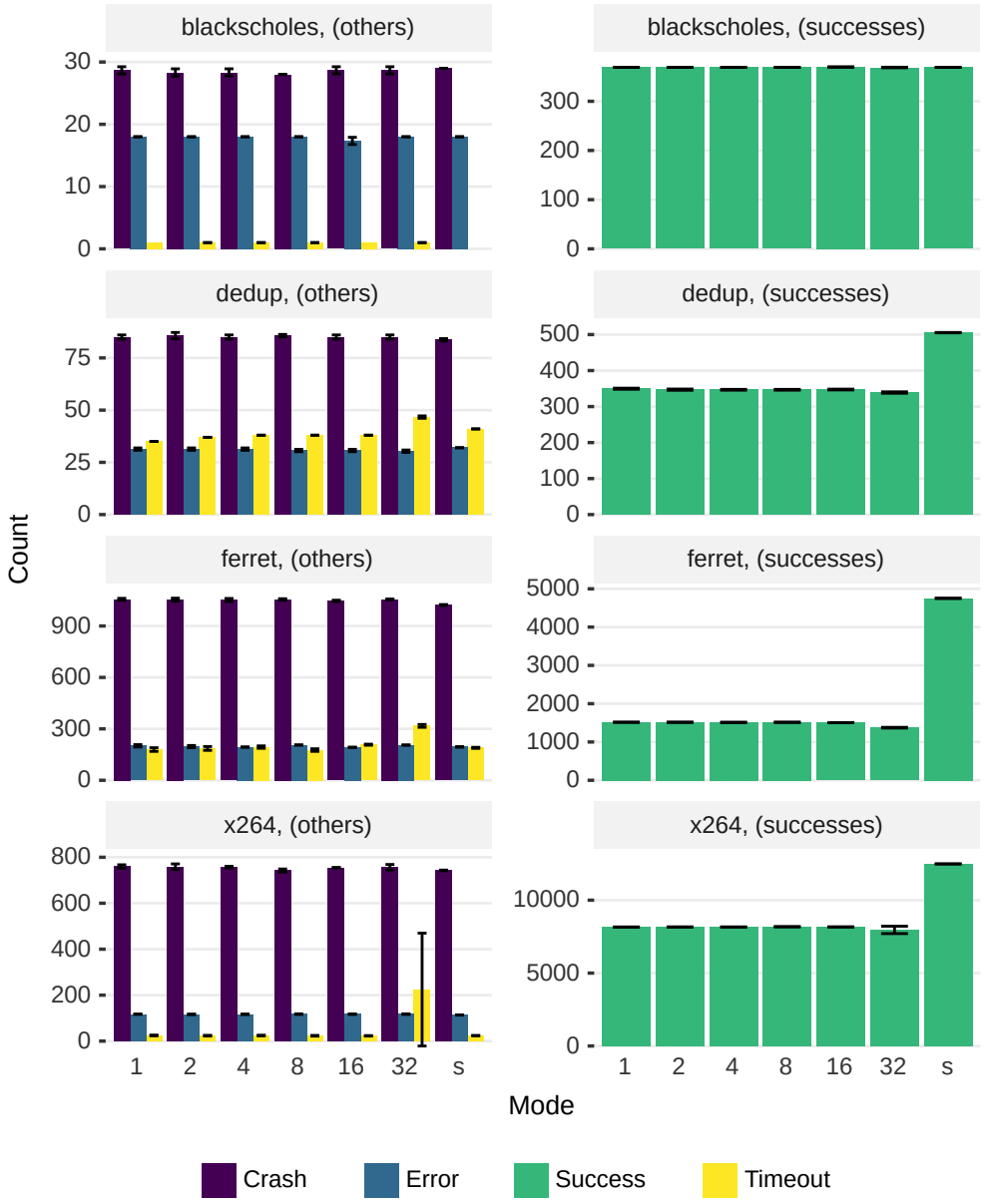


Figure 3.8: SFI test results for different modes of execution and degrees of parallelism. The x axis labels indicate the employed degree of parallelism (P_n) for FastFI execution. The "s" label indicates the sequential single version mode execution. Error bars indicate standard deviation.

degree of parallelism can be avoided by choosing a higher timeout threshold, at the cost of increased SFI test latency as we showed in the PAIN experiments.

> *SFI test results obtained with parallel* FastFI *execution remain stable if timeout threshold are sensibly chosen and the parallelism degree does not overload the host platform.*

3.5.5 RQ 4: Build Time Overhead

To investigate how large the overhead for creating integrated FastFI executables is, we build the same set of faulty versions twice: once with FastFI and once by building separate executables for each faulty version. In the latter case, we utilize incremental compilation. Therefore, for each faulty version, one compilation unit is recompiled and the final executable is linked. This is a typical approach for building faulty program versions for SFI tests. User times for building with FastFI relative to the traditional model are shown in Figure 3.9. Note that the recorded times include the application of the mutation patches and, for FastFI, code generation. FastFI offers substantially lower build times compared to the traditional approach: FastFI builds take between 7.2 % and 20.8 % of the user time required for building separate executables for each faulty version. This corresponds to a speedup between 4.8 and 13.9. For `x264`, this speedup reduces the build time from almost 6 h to 35 min. The reason for this advantage is that FastFI avoids redundant recompilation: The traditional approach incurs substantial overhead due to always recompiling entire compilation units, even though only a single function differs from the fault-free version. Since FastFI works at function granularity, it avoids this overhead by design.

> *Integrated* FastFI *executables containing all faulty versions can be built significantly faster than traditional single version executables. We achieved a best case build time speedup of 13.9×.*

3.5.6 Discussion

Our investigation of FastFI with regard to our four research questions shows that FastFI can be applied to real world software and it is effective at avoiding redundant code re-execution, enabling sequential speedups of up to 3.6 over a traditional execution model. Our results also show that FastFI enables further speedups through parallelization, which can be even further improved by using different parallelization strategies than the one implemented in our prototype.

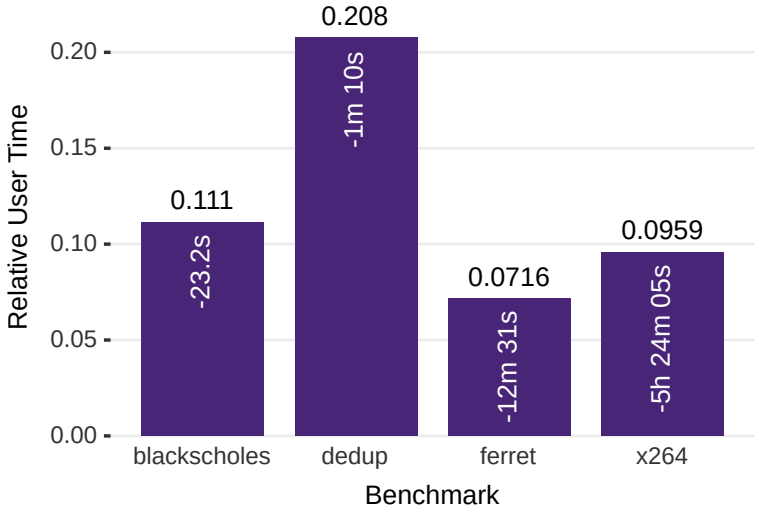


Figure 3.9: FastFI user build times relative to user build times for separate executables. The absolute reduction in overall user build times is given within the bars.

FastFI therefore enables the effective utilization of modern parallel computing hardware for SFI tests. We find that neither sequential nor parallel FastFI execution adversely affects SFI test result stability unless the degree of parallelism exceeds the available computational resources, in which case spurious timeouts commonly arise as we also observed in PAIN experiments. Such issues can be addressed by adjusting timeout thresholds at the cost of potentially higher execution latencies. Finally, our investigation shows that FastFI enables faster compilation of faulty versions due to the finer, function-level granularity our approach offers. Overall FastFI reduces latencies for both the compilation of faulty software versions and their execution.

3.5.7 Concluding Remarks

With the increasing complexity of our software, we have to conduct an exploding number of SFI experiments for assessing the dependability of said software. Therefore, we developed FastFI a novel approach that is inspired by the insights from our PAIN study and that combines several techniques to accelerate the execution of a large volume of SFI experiments. To that end, FastFI (1) avoids redundant code execution, (2) avoids the execution of "dead" faulty versions, (3) executes tests in parallel, and (4) reduces build times for faulty versions. Based on our evaluation of FastFI on benchmark programs from the PARSEC suite, we conclude that FastFI is applicable to real world software from various application domains, enables both sequential execution speedup as well as effective parallelization, and substantially

reduces build times. Relying in the insights from PAIN, FastFI experiments can be easily dine-tuned to avoid result accuracy issues.

In future work, we plan to extend FastFI in several directions. Different parallelization strategies, such as replacing the fixed chunks currently used by FastFI with work stealing, may result in improved CPU utilization and a further reduction in SFI test latencies. Our current prototype is limited to programs written in C and we are planning to support C++ as well. Moreover, we plan to extend FastFI to support concurrent software.

3.6 Related Work

In the following, we discuss prior work that is related to both our PAIN and FastFI approaches for accelerated execution of SFI tests.

3.6.1 Fault Injection (FI)

FI has been employed extensively across the whole software stack as well as at the hardware level in different scenarios, including applications in embedded, safety-critical, real-time, and operating systems. For instance, Arlat et al. [Arl+02] applied FI to a microkernel OS that is composed of OTS components. Ng and Chen [NC01] identified and fixed issues in their file cache design using FI. Others have used FI to conduct dependability benchmarking to design alternative of software components for web servers [DVM04], database systems [VM03], and operating systems [DM03; KD00]. Much work concerned with FI investigates metrological aspects to avoid drawing false conclusion about a system's dependability in the presence of many complex factor influencing FI experiments. A number of approaches have been proposed to limit the intrusiveness of FI techniques, by relying on already existing debugging mechanisms provided by the hardware [Aid+01; CMS98] and by minimizing modifications in the target software [Sto+00]. Skarin et al. [SBK10] assessed the metrological compatibility of results produced from these alternative techniques with limited intrusion. Kouwe et al. [KGT14] assessed the result distortion because of injected faults that show no effect on the system and are under-represented. As distributed systems are commonly affected by non-determinism and clock skew issues, the repeatability of FI experiments in this context has been evaluated [Cha+04; Cot+13b]. Irrera et al. [Irr+13] evaluated whether VMs can be used for FI experiments without adversely impacting certain system metrics. Although they drew a positive conclusion, the conducted experiments showed that using VMs had a noticeable impact on some of the metrics they recorded.

3.6.2 FI Test Throughput

A number of studies have advocated the potential benefits of parallelizing FI experiments [Ban+10; BC12; Han+10; Mah+12] using virtual machines [Ban+10; Han+10] or OS processes [BC12] to isolate the experiments. Although virtual machines provide execution environments with stronger isolation, the run time overheads that virtual machines incur can cause performance interferences, which can equally distort the results of fault injection experiments as we showed in our PAIN experiments. As a consequence, we chose to restrict FastFI's isolation for concurrently executing experiments to lightweight processes, which makes it a suitable approach for FI testing above the OS level.

3.6.3 Test Parallelization

As in many other areas of technology, the idea of parallel testing has been driven by emerging parallel hardware and system designs [Sta00]. Parallel test execution has been used to improve the throughput in regression testing [Kap01] and in MapReduce-based unit testing on cluster hardware [Par+09], but also to test a complex CORBA implementation across different platforms [Las05]. Other recent approaches advocated the Testing-as-a-Service (TaaS) paradigm that fits well into the Cloud computing landscape for dynamic testing [Yu+09; Yu+10] as well as for static testing (i.e., program analysis) [CBZ10; Cio+10; Mah+12; SP10].

Until recently, many approaches parallelized test executions under the assumption that these tests are independent and do not influence each other [Dua+06; Mis+07; OU10; Par+09]. This assumption has proven incorrect for a number of test suites [CMd17; Zha+14]. Newer approaches take possible test dependencies into consideration and use this information to determine which tests need to execute in sequence to prevent spurious results [Bel+15; Gam+17; LZE15]. In FastFI, concurrently executing program versions do not interfere as external resources are carefully handled by the runtime. As soon as a faulty version is selected for execution, a new process is forked to guarantee memory protection via address space isolation. Possible interference on shared persistent file storage are prevented by means of I/O redirection. Thus, the isolation across parallel SFI tests is stronger than what is commonly assumed for parallel correctness tests, but weaker than the VM-based isolation than we use with PAIN for parallel fault injections to reduce the risk of performance interference that we encountered with PAIN.

3.6.4 Avoiding Redundant Code Execution

FastFI saves execution time by avoiding redundant and unnecessary code executions. We are only aware of one work that makes a similar attempt to reduce test suite execution latency. VmVm [BK14] analyzes which data is modified by each

individual test case in a test suite and makes sure that the test suite executor only resets that part of the system state between tests, so that heavier isolation mechanisms can be avoided. The authors report an average execution time reduction of 62 %. In contrast to VmVm, FastFI avoids (a) the execution of code paths that are redundant for many tests and (b) the execution of faulty program versions, for which the fault would not get activated. These redundancies are peculiarities of FI tests and usually do not apply for other types of tests, such as unit tests targeted by VmVm. FastFI also does not attempt to reduce isolation between tests, but utilizes this isolation to safely execute tests concurrently to gain additional speed-up from parallel hardware.

3.6.5 Result Validity with Parallel Execution

Prior work that exemplified the benefits of parallel FI [Ban+10; BC12; Han+10; Mah+12], did not investigating whether parallelism affects the validity of test results as we do with PAIN as well as FastFI. Often, especially if conducted in VMs, FI experiments are assumed to be inherently independent and therefore easy to parallelize ad infinitum. However, there are possibly adverse effects due to parallelization as performance isolation between VMs cannot easily be guaranteed [Gup+06; SC09] as was also the case in some of our PAIN experiments. Software executing in different VMs can suffer from performance interference, for instance, if the host runs out of memory or the CPU is overloaded, leading to different system behavior compared to execution outside a VM that can even affect the system's security [HL13; Nov+13].

3.7 Conclusion

Due to increasingly complex software stacks and application scenarios, together with emerging SFI techniques that combine multiple faults, we have to cope with an explosion in the number of SFI experiments to be conducted for comprehensive dependability assessments of said software. The parallel execution of SFI experiments seems to be a promising approach to compensate for this large volume of required experiments. However, with the parallel execution of such experiments, the question arises whether the obtained experimental results remain stable and valid with the increasing degrees of parallelism that modern hardware enables.

Therefore, we started by assessing whether we can achieve higher experiment throughput by performing OS-level SFI experiments in parallel VMs using our PAIN framework. Moreover, we assessed if the obtained result distributions change with the introduction of parallelism. To that end, we defined measures for experiment performance and for result accuracy for evaluating. We applied our methodology to study the effects of faulty drivers in the Android OS. Our results show that

PAIN can considerably improve experiment throughput, but at the same time lead to result inaccuracies. These inaccuracies were related to the chosen degree of parallelism as well as to timeout thresholds for failure detection. In our analysis of PAIN experiments, we provide insights and guidelines that others can use to fine-tune their parallel setups and avoid the mistakes we initially made.

Based on our experience with PAIN and parallel execution in VMs, we developed FastFI, a novel SFI execution approach that relies on processes for experiment execution and accelerates the overall SFI process by (1) avoiding redundant code execution, (2) avoiding the execution of "dead" faulty versions, (3) parallelization of test execution, and (4) reducing build times for faulty versions. Applying FastFI to PARSEC applications, we achieve speedups of up to 3.6× in sequential execution and up to 26× in parallel execution. The number of executed faulty versions could be reduced by up to 52.1 %. FastFI can reduce build times to as little as 7.2 % of conventional SFI approaches. FastFI achieves these improvements while maintaining result stability and is therefore a viable approach for reducing SFI test latencies in real world settings.

4 Towards Parallel Testing for C

Testing is a time consuming activity performed during software development. In Chapter 3, we showed that the efficiency, i.e., execution latency, of SFI tests can be significantly improved by exploiting parallel hardware. But the execution of functional and correctness tests as well as the analysis of their results is also an important part of testing activities. With increasingly parallel hardware, the execution latency of a test suite strongly depends on the degree of concurrency with which test cases are executed. However, if test cases have not been designed for such concurrent execution, they may interfere, which can lead to result deviations compared to traditional sequential execution. To prevent such interferences, each test case can be provided with an isolated execution environment, but this entails performance overheads that diminish the merit of parallel testing. In this chapter, we present a large-scale analysis of the Debian Buster package repository, showing that existing test suites in C projects make limited use of parallelization. We then present an approach to (a) analyze the potential of existing C test suites for *safe concurrent execution*, i.e., result invariance compared to traditional sequential execution, and (b) execute tests concurrently with different parallelization strategies using processes or threads if it is found to be safe in step (a). To demonstrate the utility of our approach, we apply it to nine projects from the Debian software repositories and analyze the potential for concurrent execution. The contents of this chapter are, in parts verbatim, based on material from [Sch+19].

4.1 Overview

Dynamic software testing, i.e., the controlled execution of software and the comparison of its behavior against specified behavior, is widely applied to identify software defects. To obtain high test throughput and limit the influence of human error, dynamic software tests are themselves commonly implemented as software for test automation. As the amount of test code has exceeded that of the application logic by far for numerous projects [GVS17], its execution time is critical for the performance of various steps in software development and maintenance. For maintainability and selective execution, the test code is organized as collections of test cases in *test suites*. With the increasing parallelism of modern processors, test execution times can only benefit from increasing computational power if test suites are designed for concurrent execution. The total execution time of a test suite

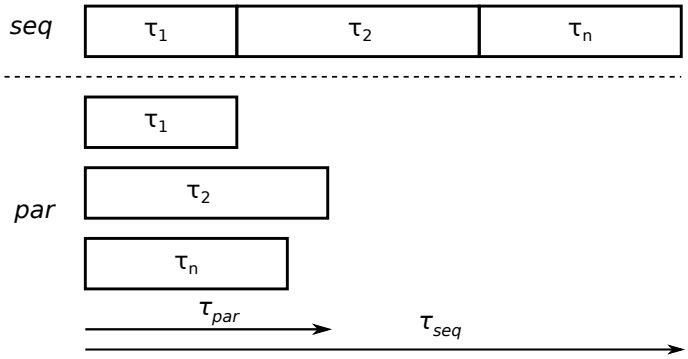


Figure 4.1: Illustration of the intended achievement. Execution time for the parallel case τ_{par} is defined by the longest executing test case, whereas for the sequential case τ_{seq} it is defined by the sum of all test case execution times.

consisting of test cases $t_1 \dots t_n$ with execution times $\tau_1 \dots \tau_n$ would be reduced from $\sum_{i=1}^{n} \tau_i$ in the sequential case to the execution of the longest running test $\max(\{\tau_1 \dots \tau_n\})$, as illustrated by Figure 4.1, if two conditions hold: (a) sufficient parallel processing units are available, and (b) all tests in a test suite are *independent*.

Unfortunately, this assumption of test case independence within a test suite has proven problematic [Zha+14]. Even sequential executions of a test suite can lead to differing test case results across different permutations of their execution order. The major root causes behind test dependencies found in existing software projects have been identified as (a) shared global memory and (b) shared files [Zha+14]. While the former has been identified as the most common reason for test dependencies (62.7 % of all dependencies analyzed in [Zha+14]), it is only problematic if test executions share the same memory address space. Isolating tests in individual processes would, therefore, solve a substantial portion of the problem, but reportedly induces significant overheads on test executions [BK14]. As shared files affect any tests operating on the same file system, file dependencies need to be identified irrespective of address space isolation.

In this chapter, we explore several implementation alternatives (with different degrees of memory isolation) to achieve safe parallel executions of existing sequential test suites for projects written in C. By *safe* parallel execution, we mean that the results of test cases executed in parallel *cannot* differ from the results of their original sequential execution order. We focus our work on C, the predominant language in the Debian main package repository (as we will show in Section 4.3). C also features the second highest test count across projects hosted on GitHub according to a study of Kochhar et al. [Koc+13]. To check if tests can interfere in parallel execution, we implemented two static analyses on LLVM IR, the intermediate representation used by the LLVM compiler infrastructure [LA04]. This means that our analysis could also be applied to software written in other languages than

C if a suitable LLVM front end exists. The decision to focus on existing test suites is motivated by the large amount of existing sequential test code that is shipping with widely used software.

As a brief summary, this chapter presents the following contributions.

- An analysis of Debian Buster's main package repository showing that the majority of code contained in the packages is written in C, that no test framework dominates test implementations for C packages, and that few test suite implementations benefit from concurrent execution in Section 4.3.
- We develop automated static analyses for C programs to identify test case interdependencies on files and shared global data in order to identify which parts of a test suite can safely execute in parallel in Section 4.4.
- We develop a test harness to use this information for safely executing tests in parallel and explore the trade-off between address space isolation and parallel test suite performance in different parallelization alternatives using processes and threads for nine Debian source packages.
- We present the results of an in-depth analysis of nine software projects from Debian Buster, for which we parallelize test execution using our dependency analyses and test harness in Sections 4.5 and 4.6. Our results show that test suites in C can benefit from even modest degrees of parallelism provided by virtually every desktop or server hardware configuration, that threads do not perform significantly better than processes, and that our test harness (and likely any specialized test tool) outperforms generic automation tools like `make`.

4.2 Related Work

The goal of our work is to assess if the concurrent execution of tests in C projects can achieve better latencies without compromising test outcomes. Articles related to our work fall in three categories:

1) Articles with the same objective and mechanism, i.e., the concurrent execution of tests for latency improvement.

2) Articles with the same objective, but different mechanisms, i.e., latency improvements of test execution by other means.

3) Articles with similar mechanisms, i.e., test interference detection, but with different objectives.

In summary, only one existing approach (VmVm [BK14]) does not require the execution of tests. Parallelization approaches based on dynamic analyses suffer from the need to execute the test suite at least once. After the test suite has been executed once, the test results are known and there is no benefit in obtaining the same results again, no matter with which run time improvement. Hence, dynamic approaches are only useful if there is a possibility for test results to differ across repeated executions and if that is the case, the same effects that cause the test results to differ may alter the analysis results that the parallelization is based on as well. Therefore, we need to rely on static analyses to detect test interferences for enabling safe concurrent execution. As we cannot reuse VmVm's static analysis because it operates on Java code, we develop static analyses of accesses to global variables and shared files for C programs as LLVM compiler passes.

4.2.1 Concurrent Test Execution for Latency Improvement

Early approaches for concurrent test executions [Dua+06; Mis+07; OU10; Par+09] *assume* test cases to be independent and do not analyze if their parallel execution possibly alters test results. As test dependencies were found to affect permutations of test sequences [LZE15; Zha+14], newer approaches address the possibility of test dependencies.

ElectricTest [Bel+15] identifies dependencies in Java tests to determine which tests need to execute in sequence to prevent spurious results. The dependencies are derived from execution traces of shared resource accesses, which are gathered during test execution. Lam et al. [LZE15] assess the impact of dynamically detected test dependencies in Java projects on test parallelizability, achieving execution speedups between 1.02× and 7.14× depending on the project and number of CPUs.

CUT [Gam+17] executes unit tests in parallel and isolates them in separate virtual machines or Docker containers to ensure that concurrently executing tests *cannot* interfere. CUT relies on external input in the form of a directed acyclic dependency graph, which can be provided by analyses like those presented in this chapter.

O!Snap [GGZ17] uses VM snapshots to speed up test execution. To avoid missing libraries or setup steps for running the tests, O!Snap analyzes dependencies on the software package level. Our approach is orthogonal, as it targets concurrency of tests within a package, as opposed to concurrency across packages.

Candido et al. [CMd17] investigate how commonly concurrent test executions are used in open source projects. Their results show that only 13 out of 110 investigated Java projects execute tests concurrently. The authors experimentally assess the speedup (up to 75.9×) and the rate of spurious test failures (up to 96.3 %) of naive parallelization that ignores dependencies, emphasizing the importance of dependency analyses for test parallelization. Our complementary study for C

projects in the Debian Buster repository confirms the finding that few projects can benefit from parallel testing out of the box.

4.2.2 Improving Test Latencies without Concurrency

The improvement of test execution latencies is a main driver behind entire research directions within the testing community, such as test selection or prioritization. We found only one project that, similar to parallelization, achieves latency improvements without omission of tests. VmVm [BK14] reduces the execution latency of sequential test suites by replacing costly per-test initialization and termination routines with lightweight reset routines that are sufficient to provide non-interference across consecutive tests. To identify which part of the software under test's (SUT) state needs to be reset, VmVm uses a static analysis to identify heap memory that is possibly accessed by multiple tests.

4.2.3 Test Interference Detection

Another reason for analyzing test interdependencies is the identification of bugs in test code. If individual tests are supposed to be independent from each other, any interdependency indicates a bug.

Muşlu et al. [MSW11] propose to execute tests in isolation to reveal dependencies on other tests and report an actual bug in Apache Commons CLI using this technique.

DTDetector [Zha+14] permutes the execution order of Java test suites to identify unintended test dependencies via static fields. To keep the execution overhead tractable, DTDetector samples permutations using different algorithms, one of which uses test (in-)dependence information to filter permutations that cannot reveal test dependencies. To gather dependency information, DTDetector executes each test once in isolation.

PraDet [GBZ18] detects manifest test dependencies with a similar approach as DTDetector, but reduces false positives by using an enhanced version of ElectricTest's [Bel+15] dependency detection.

PolDet [Gyo+15] detects *state pollutions* of shared state across Java tests by identifying shared heap memory at run time and tracking accesses to the identified regions. PolDet also tracks modifications to files, but relies on user input for identifying which files are relevant and need to be tracked.

4.3 Empirical Study: C Software in Debian Buster

In our literature review (Section 4.2), we made the observation that all existing work on test dependencies is focused on Java projects. While we do not speculate

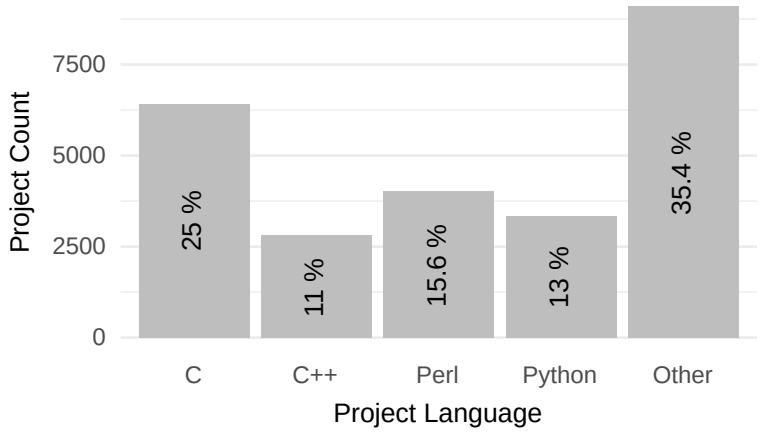


Figure 4.2: Number of packages in Debian Buster by their dominant language. The absolute number is on the Y-axis, the relative number within the bars. Languages below 5 % are grouped together as *Other*.

about the reason, we needed to confirm that it is *not* because testing of C code is an irrelevant problem. For this purpose, we analyzed the entire main package repository of the upcoming "Buster" release (version 10) of the Debian Linux Distribution [Deb18] with three major objectives:

1) Assessing the amount of C code in the repository compared to other languages to confirm the relevance of our work

2) Assessing which test frameworks are most widely used to test C code in the repository

3) Assessing to which extent parallel execution is able to improve test performance for packages that already support it to some degree out-of-the-box

4.3.1 Programming Languages in the Debian Ecosystem

We started by downloading the source package index for "Buster" [Deb17], which lists the source code packages and their download locations. We then downloaded and unpacked all 25 684 source packages available in Buster. To determine both the programming languages and the amount of source lines of code (SLOC) per package, we used `cloc` [Dan18] and excluded markup languages such as XML or JSON, as these are used for describing data rather than executable code.

Figure 4.2 shows the total number of packages by their predominant language, i.e., the languages that contribute most SLOC to the respective packages, and the relative contribution of each language to the entire repository. With almost 25 %, C is the most prominent language across all packages. To affirm that this finding

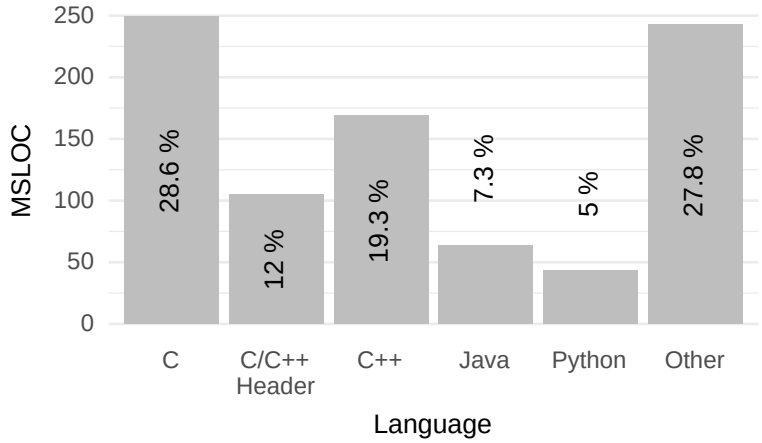


Figure 4.3: Millions of source lines of code (MSLOC) contained in the Debian Buster repository by language. The absolute number is on the Y-axis, the relative number within the bars. Languages below 5 % are grouped together as *Other*.

is not biased by differing amounts of code in the packages, we also accumulated the SLOC number per language across all packages as shown in Figure 4.3. With around 250 million SLOC, more than 28 % of the total code in the repository is written in C. This number excludes code in header files, as `cloc` cannot distinguish whether they belong to C or C++ code, and is, thus, a conservative estimate.

C is the dominant programming language in the Debian package ecosystem.

4.3.2 Test Frameworks

To analyze the use of test frameworks, we scanned the downloaded sources for JUnit usage in the case of Java and for the presence of typical files and directives of 34 different freely available test frameworks[1] in the case of C. Figure 4.4 on the next page summarizes our findings. We found that, with less than 5.5 %, only few C projects make use of any of the test frameworks. This is in strong contrast to the situation for Java, where over 65 % of projects use the de facto standard JUnit for testing.

[1]For C projects we search for: `libcbdd`, `AceUnit`, `AutomatedTestingFramework`, `Autounit`, `Cgreen`, `CHEAT`, `Check`, `Cmocka`, `Cmockery`, `CppUTest`, `Criterion`, `CU`, `CTest`, `CUnit`, `CuTest`, `Cutter`, `EmbeddedUnit`, `FCTX`, `GLibTesting`, `GUnit`, `lcut`, `LibU`, `MinUnit`, `Mut`, `NovaProva`, `OpMock`, `RCUNIT`, `SeaTest`, `Sput`, `TestDept`, `TFUnitTest`, `Unity`, `tinytest`, `xTests`

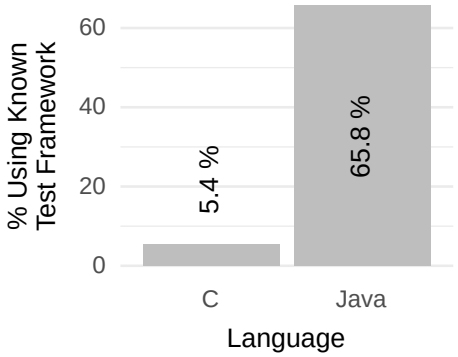


Figure 4.4: Usage of known test frameworks in C- vs. Java-dominated packages. The Y-axis represents the percentage of packages using a framework relative to all packages with the respective dominant language.

No test framework is commonly adopted for C software in the Debian package repositories.

4.3.3 Test Parallelization

To detect if packages can benefit from parallel test execution, we identify all packages that show indications for the presence of any tests by scanning for file and folder names that include "test" as substring. By invoking typical build and test execution targets of GNU `make`[2], we then build each of these packages, execute their test suites, and measure the tests' execution times for varying degrees of execution parallelism specified via `make`'s `-j` flag. We repeat our time measurements three times per configuration to account for possible variations due to factors that are not under our control.

Out of 6419 C packages in the repository, we identified 1617 to show indications for the presence of tests. Out of these, 627 completed our measurement without failure for all three runs. Most packages that failed did so in a consistent way for all three runs (99.2 %). A remaining 8 packages exhibit flaky build or test behavior. Half of them had test failures in the parallel case, despite successful sequential test executions. We also observed such behavior among 10 packages that failed consistently in each of the three repetitions.

From the 627 non-failing packages, only 177 (28 %) had shorter test execution times for the parallel case in all runs. 261 packages (41.3 %) had equal or longer test execution times compared to sequential execution. The remaining packages

[2]We also invoke typically found configuration steps such as `autoconf` or `configure` and try different `make` targets for executing tests such as `check` or `test`.

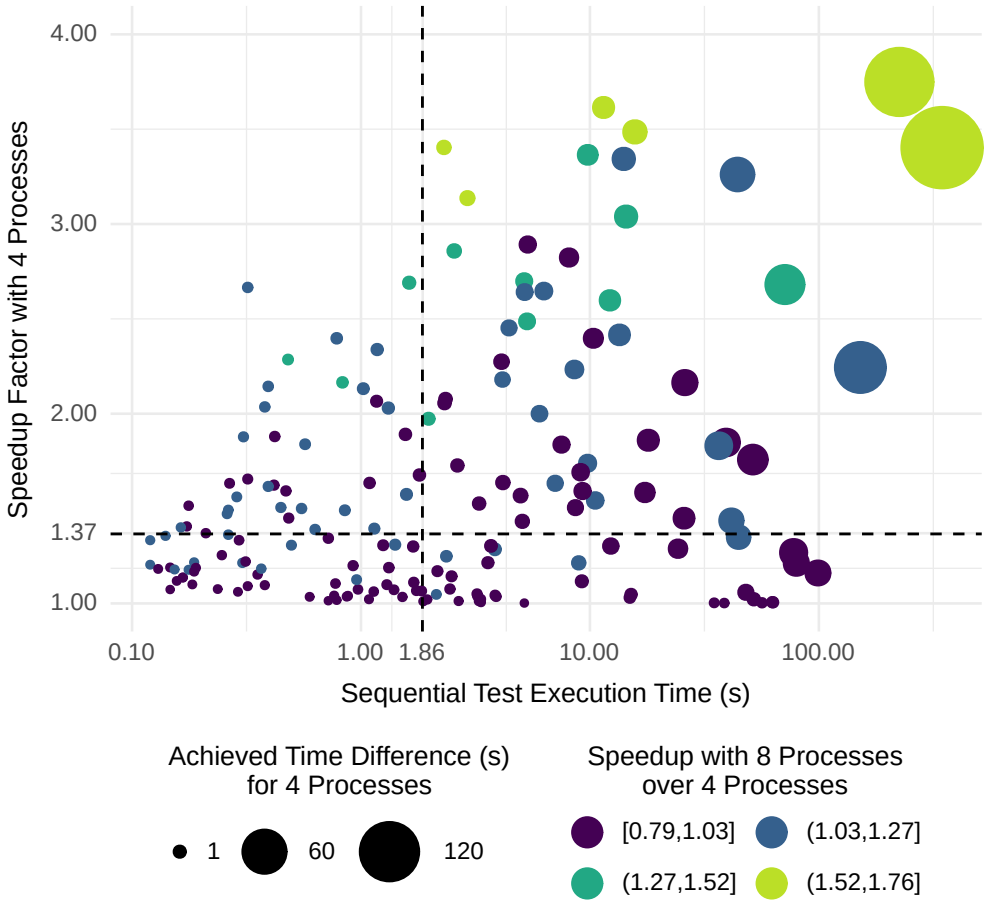


Figure 4.5: Achievable test speedup for C software packages in Debian Buster. The dashed lines indicate the median sequential test execution time and the median speedup achieved with 4 processes. The size of the bubbles indicates the time difference between sequential and 4-fold parallel execution. The color coding illustrates additional speedup achievable by 8-fold parallelism.

did not yield clear results, with parallel test performance sometimes exceeding the sequential case and sometimes vice versa.

The achieved test execution time speedup factors for the 177 benefiting packages are shown in the bubble plot in Figure 4.5. From the plotted data we observe that the degree by which projects benefit from parallel test execution varies greatly. While it is not surprising that longer sequential execution times (on the x axis) tend to coincide with bigger time savings (bubble size), it is remarkable that even projects with short test suite execution times between 250 ms and 1 s can achieve speedups well above the median of 1.37.

If the degree of parallelism for test execution is increased from 4 to 8, we observe only modest additional speedup, as indicated by the bubble colors, for the majority

of the projects. Almost 60 % fall in the lowest category and 30 % in the second lowest. More than half of the projects in the lowest category have a speedup of 1 or less, i.e., they do not benefit from increased parallelization.

Test parallelization via command line flags works for less than 39 % of C packages that use `make` *for test execution. Most of these packages do not deterministically benefit from 4-fold parallel test execution. Out of those that do, few can benefit from further increased parallelism.*

4.3.4 Threats to Validity

We do not claim that the results from our study apply for other ecosystems. With Debian, our study targets a large ecosystem that forms the basis of many production software stacks [Deb18]. The downside of this choice, which guarantees practical relevance, are potential inaccuracies in our analyses resulting from the need to scale them to an ecosystem of significant size and projects with limited support for automated analyses. Our analysis of dominant languages relies on `cloc`'s accuracy, which is widely used for SLOC counting. Our analysis of test frameworks depends on the list of frameworks we searched for in the projects and the accuracy of our search heuristics. Similarly, the conclusions from our test run time analysis may depend on our build and test automation. Our conclusions are drawn from three repetitions of the run time analysis. We have used the coefficient of variance as a rough measure to detect massive instabilities, which we only found in one case of averaged time differences for 8-fold parallelism and which we excluded from the analysis. The exit codes observed were stable across the three conducted runs in more than 99 % of the cases, which adds to our confidence in the absence of massive deviations from the reported results.

4.4 Safe Concurrent Testing for C

Our empirical study has shown that only a small fraction of those C projects in Debian that invoke tests via `make` benefit from parallel test execution. In the following we present an approach to (1) assist C developers with the implementation and maintenance of concurrent test suites and (2) enable safe concurrent test executions for legacy test suites that have been designed for sequential execution.

Figure 4.6 on the next page gives an overview of our approach. The three phases of preparation, analysis, and safe concurrent test execution are discussed in the following subsections.

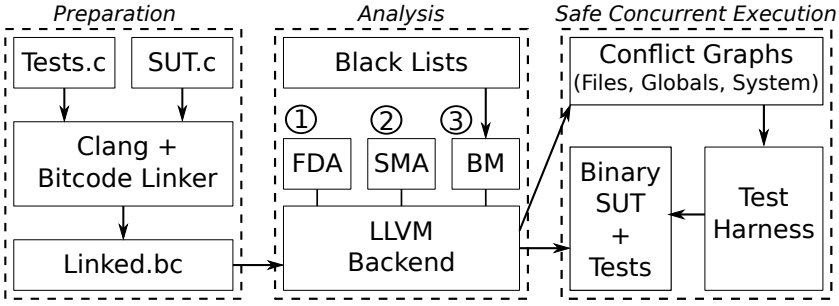


Figure 4.6: To prepare for our analyses detailed in this section, the tests and the SUT have to be compiled to bitcode and linked. After running our analyses ① – ③, which are implemented as LLVM compiler passes, we obtain information on potential test conflicts. These are leveraged by our test harness to derive safe parallel test schedules.

4.4.1 Preparation

We implement our analyses as LLVM optimizer passes performing a whole-program analysis on the tests and the SUT. For this purpose, we require the tests and the SUT to be compiled to LLVM bitcode and linked together. Everything that is not linked in at the point at which our passes run is deemed external and any test inter-dependencies due to external resources must be addressed via a *blacklisting mechanism* discussed in Section 4.4.2.

Our analyses assume test cases to be *self-contained*, i.e., not to rely on external inputs. External inputs are either generated by human testers or by external test automation tools written in other languages. If the test suite relies on human input, its potential performance gain from automated parallelization is limited. If input data is generated by tools written in other languages, those parts of the test harness would require an analysis engine for those languages. If any input generating code can be linked with the LLVM-IR of the tests, our approach can include it in the analysis.

4.4.2 Detecting Potential Test Interference

Concurrent test executions can interfere if two or more test cases access the same data, at least one such access is modifying that data, and the test outcome of at least one other test depends on that data. Which data is shared among concurrent tests depends on their execution environment. Concurrently executing tests in separate *processes* (as in the case of `make` in Section 4.3) share the same operating system state (e.g., system wide configurations like the locale) and in particular the same file system, but not the same memory. Dependencies on shared memory only affect concurrent tests if they execute as threads within the same process context. We developed separate static analyses to detect potential dependencies (due to

global variables or file system usage) in a given test suite, because of these different parallelization strategies they enable.

We chose a static approach for analyzing potential dependencies over a dynamic approach since static analyses have the advantage that the analyzed tests do not need to be executed. A dynamic analysis would already produce the desired test results, limiting the utility of the approach to cases for which a repeated execution of the same tests in the same configuration is desirable. Static analyses can be integrated into the software build process which ensures that the used dependency information always matches that of the produced test executables. This integration is especially useful if a software project has many build-time configuration options, which may influence test dependencies.

Analysis ①: File Dependency Analysis (FDA)

To detect file dependencies, our analysis first checks whether certain known functions that are used to interact with the file system, such as `fopen`, are reachable from a test case by constructing the static call graph for the SUT and traversing it for each test case's SUT invocations. Then, for each call site of such a function that is reachable from at least one test case, we traverse use-definition chains to determine which (constant) file names may be passed to the function. A test case t may access a file f if a call site of a file processing function is reachable from t and f is a reaching definition for a function argument at that call site. We use the same technique for mode arguments to distinguish read-only accesses from writing accesses. The resulting file read and write sets $F_r(t)$ and $F_w(t)$ for each test case t can be used to detect dependencies between any pair of test cases and we construct an undirected test case conflict graph $C_F = (V, E)$ as follows:

- For each test case, we add a corresponding vertex to V.
- For each pair of vertices $t_i, t_j \in V$, we add an edge to E iff $(F_r(t_i) \cap F_w(t_j)) \cup (F_r(t_j) \cap F_w(t_i)) \cup (F_w(t_i) \cap F_w(t_j)) \neq \emptyset$, i.e., when there is a possibility of accesses to the same file including at least one write operation.

Analysis ②: Shared Memory Analysis (SMA)

Analogous to Section 4.4.2, we construct the static call graph of the SUT and the tests. We then follow the definition-use chains of all global variables of the SUT, as well as function arguments in cases where global variable addresses are passed as parameters, to identify which of them may be read or written in which test case. We consider it sufficient to focus on global variables, because (1) global variables are implicit heap allocations and shared among threads, (2) function-local variables are allocated on the stack and are, thus, thread-local and not shared among concurrent

threads, (3) to share explicitly allocated heap data (e.g., via `malloc`), threads need to communicate its addresses, which is only possible via previously shared memory.

Our analysis does not identify shared memory accesses to hard-coded constant-value addresses. Such accesses constitute a severe risk to memory safety and must be considered bad practice for commodity systems. For embedded systems there may be cases of software containing hard coded addresses. For these scenarios, our analysis would need to be augmented with a (straight-forward) mechanism to analyze constant propagation.

The result of our analysis is a mapping that assigns to each function f in the module its read and write sets of global variables $G_r(f)$ and $G_w(f)$. A test case t may read or write a global variable g if any of the functions reachable from that test case according to the static call graph may read or write g. Thus, the set of global variables that may be read (or written) during execution of t can be computed as $G_r(t) = \bigcup G_r(f_i)$ and $G_w(t) = \bigcup G_w(f_i)$ of all functions f_i reachable from t. The resulting read and write sets for each test case can then be used to detect dependencies between any pair of test cases by constructing the conflict graph C_G for global variables analogous to C_F above.

Analysis ③: Blacklisting Mechanism (BM)

As previously mentioned, we rely on a blacklisting mechanism to model test dependencies on shared system resources besides files and memory. This mechanism takes a list of functions as input that access such shared resources, along with additional information whether the access is reading or writing the shared resource. We analyze the test cases and the SUT for call sites of these functions and create read and write sets of shared resources for each function in the module, analogous to how we handle global variables. We then reuse the static call graph constructed during the shared memory analysis to determine which of the identified call sites can be invoked during test execution. The resulting conflicts are added to C_G, thereby effectively modeling them as global variables.

4.4.3 Concurrent Test Execution

The orchestration of test executions is generally implemented in some *test harness*. As we found in our empirical study in Section 4.3, C projects frequently use the general purpose build automation tool `make` for this purpose. We implement a custom test harness in our work to achieve the concurrent execution of test cases. Our test harness supports different parallelization strategies that make use of the dependency information extracted by our static analyses to prevent test interferences.

In general, there are two options for concurrent test executions, which differ in their risk of interfering test executions and their run time overhead: (a) executing

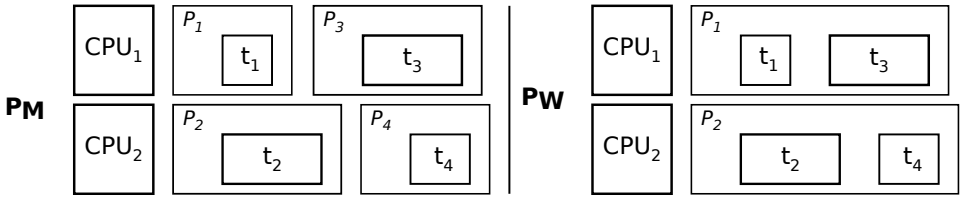


Figure 4.7: Multi-Process strategies P_M and P_W: For P_M a clean process is started for each test, whereas for P_W the repeated process initialization overhead is saved by reusing processes as long as tests do not have dependencies.

tests in parallel, isolated *processes* or (b) executing tests in parallel *threads* without memory isolation.

Option (a) provides isolated address spaces, which eliminates memory interferences for parallel tests. Option (b) does not provide this isolation, but offers lower overhead compared to (a) since thread management operations do not have to create/switch address spaces for isolation. However, since option (b) lacks program state isolation for each test, all tests must be analyzed for their interference potential before they can be run in parallel. For either option we implement two execution strategies in our test harness.

Multi-Process Strategies

Our first strategy merges all test cases into one program and forks a new process for every test case (P_M). The only difference of this harness compared to `make` is that our implementation does not schedule two tests for concurrent execution if they have file dependencies, as this may lead to deviating test results. The maximum number of processes in P_M is configurable to prevent resource contention from adversely affecting test suite execution times, e.g., when the number of processes is much larger than the number of CPUs in the system.

The other option for test parallelization with processes is a *worker model* that forks a fixed number of processes, each of which executes several tests in sequence (P_W). This option avoids spawning new processes (similar to VmVm [BK14]) when sequential tests do not have dependencies and, thus, cannot interfere. Tests with file dependencies cannot execute in parallel, as previously explained in the discussion of P_M, and cannot execute sequentially within the same worker process if they have dependencies on common globals.

Figure 4.7 illustrates the difference between P_M and P_W in an example of four independent tests $t_1 \ldots t_4$ executing on two processing units CPU_1 and CPU_2. For P_W, two processes P_1 and P_2 are spawned, whereas a new process is created for each test in P_M.

Multi-Thread Strategies

We employ two multi-thread strategies T_M and T_W analogous to P_M and P_W: T_M creates a new thread for each test case and T_W uses worker threads. In addition to the dependency restrictions described for processes, threads cannot be executed concurrently or within the same worker thread if they have dependencies on global variables. This restriction does not apply for processes, as they execute within separate address spaces and do not have access to other processes' global variables.

Multi-threaded strategies, therefore, require both dependency analyses, but are expected to outperform their multi-process counterparts in terms of test execution times, because of the lower overhead for thread creation and context switching.

4.4.4 Scheduling Concurrent Test Execution

We use C_G and C_F to schedule safe, concurrent test execution according to the four parallelization strategies discussed above. For P_M, scheduling relies only on C_F. P_W, T_M and T_W all require both C_F and C_G. We use C_F to partition the set of test cases as follows: We greedily pick and remove maximal independent subsets of test cases I_i from C_F until C_F is empty. For P_M, these sets are directly used for concurrent test execution: Test cases from the same set are executed concurrently in different processes at the chosen degree of parallelism. Different sets are handled sequentially, and test cases from different sets are never executed concurrently. In the other cases (P_W, T_M, T_W), we extract for each I_i the corresponding induced subgraph from C_G. The result is a set of conflict graphs C_G^i that encode potential memory and environment conflicts among tests that do not have file conflicts. These graphs are then used to identify sets of independent tests that can safely execute concurrently (respectively, sequentially within the same process), analogous to how C_F is used in the case of potential file conflicts. By greedily constructing independent sets from the vertices in each C_G^i, our test harness avoids executing conflicting test cases in parallel in different threads of the same process (T_M, T_W) or sequentially within the same worker (P_W, T_W).

4.5 Evaluation

In the following, we evaluate our approach by applying it to 9 real world software projects. We investigate the following four research questions:

RQ 1 What are the steps involved to transmute legacy tests suites to enable the application of our analyses as well as the subsequent concurrent test suite execution with our test harness and how much manual effort is required?

Table 4.1: Evaluated Software Projects: Each project is listed with the amount of C code, the number of all and analyzed test cases, the run time (in s) of the longest running test case, and the sequential test suite execution time with both `make` and P_M.

Name	Size	Test Cases			Seq. Time (s)	
	(C SLOC)	Total	Analyzed	Longest	Make	P_M
`gnulib`	204486	1130	908	4.0	23.5	12.0
`libbsd`	7182	16	12	55.9	56.2	55.9
`libesedb`	211882	22	22	<1ms	1.0	0.01
`libgetdata`	96532	1649	1637	9.5	52.5	34.1
`librabbitmq`	9833	6	6	<1ms	0.1	0.01
`libsodium`	26123	65	65	1.1	5.4	3.9
`litl`	2403	16	10	4.0	7.3	7.0
`openssl`	244048	548	29	0.9	2.8	2.6
`sngrep`	10381	10	10	1.8	11.6	11.3

RQ 2 What dependencies do our files and globals analyses detect and do they originate in test code or in the core project code itself?

RQ 3 How high are the achieved speedups for parallel test suite execution with `make` and our test harness, and do multi-thread execution strategies achieve higher speedups than multi-process strategies?

RQ 4 How much execution time overhead do the proposed files and globals dependency analyses impose and does the overhead amortize with the achieved speedup from parallel execution?

4.5.1 Experimental Setup

We start with a brief description of our software project selection and how we performed our experiments.

Software Project Selection

We investigate our research questions by applying our approach to 9 real world software projects that are developed in the C programming language and that are included in the Debian software repository and, therefore, available on virtually every Debian-based OS installation. We selected the projects to cover a large range of project sizes, test suite sizes, and sequential test execution times, as shown in Table 4.1.

Experiment Execution

We ran our files and globals dependency analyses on each of the 9 selected projects and recorded the resulting dependency graphs. We report the detected dependency sources and the average execution times of 30 repeated analysis runs in Table 4.2 on the following page. We executed the test suites of the 9 projects in 30 experiment configurations, namely at 6 different degrees of parallelism (1, 2, 4, 8, 16 and 32) and in 5 different execution modes (`make`, P_M, P_W, T_M, and T_W), to assess how test suite execution latencies and achievable speedups change. We repeated these experiments 30 times and discuss mean values throughout this section. With 30 experiment configurations, 30 repetitions, and 9 projects, we performed a total of 8100 experiments. To achieve a fair comparison, we executed the reduced test suites (as discussed later in Section 4.5.2) for each execution mode. It is important to note that the actual test suite results did not deviate between sequential `make` and our multi-process and multi-thread execution modes, i.e. we observed the same test results with our test harness as for sequential `make`.

Execution Environment

We conducted all our experiments on a machine with Debian Buster as operating system with a Linux 4.17 (x86_64) kernel. The machine is equipped with an AMD Ryzen 7 1700X CPU with 8 physical and 16 logical cores running at 3.40 GHz. The machine has 32 GiB RAM of main memory and a 1 TiB SSD as storage medium.

4.5.2 RQ 1: Transmutation of Legacy Tests

To answer RQ 1, we report how we prepared the test suites of the 9 evaluated projects (cf. Section 4.4.1) and which manual and automated steps were involved.

First, we manually identify the test suite and its test cases in the projects source code. We exclude tests that rely on external tools or scripts written in languages other than C as these are not accessible to our analysis, as well as tests that deterministically fail in our execution environment or rely on external inputs (e.g., network or human input). To allow a fair comparison between process-based and thread-based parallelization strategies, we also remove tests that cannot be executed together within the same process by their very nature, e.g., because they close standard file descriptors such as `stdout` or otherwise corrupt their environment (e.g., sending process termination signals). We document the original number (*Total*) and the number of test cases included in our study (*Analyzed*) in Table 4.1 on the preceding page. Moreover, we verify that each test has its own unique entry point to avoid naming collisions when merging them for analysis. We integrated an automated, semantics preserving source code transformation with Coccinelle [INR18; Pad+08] in our tool chain that handles the common case

Table 4.2: Preparation and Analysis Results: Column *Diffstat* lists the amount of required manual code changes (lines added/removed/changed). Columns *Files* and *Globals* list the number of conflict inducing files and globals found in total and inside test code. Column *BL* lists the number of conflict inducing virtual globals created by blacklisting of external functions. The analysis time columns list the mean time (over 30 runs) required to find these conflicts.

Name	Diffstat	Files		Globals			Analysis Time (ms)	
	+/-/!	Tests	Total	Tests	BL	Total	Files	Globals
`gnulib`	130/0/65	5	5	5	4	15	86.35	556.35
`libbsd`	70/0/15	0	0	0	1	1	0.46	1.53
`libesedb`	6/1971/60	0	0	0	0	0	5.10	5.92
`libgetdata`	6253/875/264	36	36	1	0	4	15911.71	1106.98
`librabbitmq`	4/0/0	0	0	0	0	0	0.43	1.20
`libsodium`	80/0/4	0	0	0	0	9	4.74	80.64
`litl`	90/1/8	1	1	0	0	0	0.94	1.69
`openssl`	83/0/9	0	0	0	0	88	28.57	47.53
`sngrep`	708/0/16	0	0	0	0	0	0.87	1.97

of each test having its own `main` function by creating unique function names. In cases where a `#include` directive is used to share code for the `main` function (`libgetdata`, `sngrep`), we physically resolve the include before applying Coccinelle, i.e. we directly insert the contents of the included file. Further manual and semi-automated steps are sometimes required to allow Coccinelle to correctly parse and process the C code. For instance, we had to resolve some preprocessor macros, either manually or using the `unifdef` utility (`libesedb`, `libgetdata`, `openssl`).

Next, we adapt the project's build system to produce a single bitcode file (for analysis) and a single shared object file (for test execution with our test harness), both containing the library and test code, for which we developed general purpose scripts. To enable the linking into one file, we had to manually change the declarations of some global symbols to `static` to prevent name collisions as C does not support namespaces (`gnulib`, `libgetdata`, `litl`, `sngrep`), which means that each globally visible symbol must be uniquely named. We then apply our analyses to assess the parallelization potential of the test suite. We use the diagnostics output of our analyses, including a list of reachable external functions, to construct a blacklist (cf. Section 4.4.2) if necessary.

To allow the execution with our test harness, the assertion logic used in the tests needs to be adapted to communicate test outcomes to our test harness. To that end, we manually changed assertion macro definitions and implemented C headers to replace functions like `exit` or `abort`, which both terminate process execution and

are often found as part of assertion logic in test suites to check test outcomes, to support execution modes other than P_M.

Of the 9 projects, only 2 (`gnulib`, `libbsd`) required blacklisting for external functions. However, manual and semi-automated code modifications are usually required before our approach can be fully applied. Table 4.2 on the facing page reports the total amount of *textual* code modifications for each considered project as diff statistic (number of added, removed, and modified text lines) from the `diffstat` utility (*Diffstat* column). Apart from `libesedb`, `libgetdata`, and `sngrep`, fewer than 200 text lines were touched. The higher number of changes for the three projects is due to the manual resolution of includes and preprocessor macros as discussed above, which is a straightforward mechanical task, but touches many source code lines. Overall, we were able to convert the test suites in a matter of a few days for each project, with the exception of `gnulib` and `openssl`, which took longer as `openssl`'s test suite makes heavy use of the Perl scripting language and `gnulib` includes many tests that touch low level system functionality such as raw file descriptors and process management, which is the reason why we had to exclude a higher number of tests for those projects. In general, we expect that developers with intimate knowledge of a project and its tests could perform the conversion task considerably faster than we were able to.

> *Porting legacy test suites to our approach is feasible with reasonable manual effort and minor code modifications to the original test suites in most cases.*

4.5.3 RQ 2: Dependencies

To assess which kinds of dependencies exist between different test cases and where these dependencies originate, we examine the results of our files and globals dependency analyses. Table 4.2 on the preceding page reports the number of conflict inducing files and globals found for each of the studied projects.

We find file dependencies for three projects. For `gnulib`, the detected dependencies correspond to files that are in fact accessed during test execution but these accesses are benign (e.g., accesses to `/dev/null`, or attempts to open a non-existent file). Our analysis could be enhanced with a whitelist to account for such benign paths. We find substantially more conflicts for `libgetdata` as there is a small set of hard-coded common file names used in virtually all test cases. This prevents concurrent execution, for our approach as well as for the `make`-based execution supported by `libgetdata`. In fact, `libgetdata`'s `make`-based test execution always enforces fully sequential test suite execution, which will become more apparent in Section 4.5.4. If we attempt to concurrently execute `libgetdata`'s tests while ignoring these dependencies, we observe failing and hanging tests and in general

flaky results across repeated test executions. For `litl`, we detect one file-based conflict between two tests, in which both tests access the same file. Ignoring this dependency causes flaky behavior in parallel `make`-based test execution. Since all file dependencies we detect originate in test code, only the test suites would require modifications to remove them and enable further parallelization.

Globals dependencies are more common than file dependencies among the studied projects, and we detect them in five projects. Unlike file dependencies, most of them originate in the core project code itself rather than in test code. Such conflicts in the project code itself result from the use of global variables that are used in project code that is reachable from more than one test case. We find globals dependencies in only two test suites, namely in `gnulib` and `libgetdata`. In both cases, several tests declare their own versions of global variables using the same names (and types), which induces potential conflicts when we link several tests together. We also observe conflicts in `gnulib` and `libbsd` resulting from our blacklisting mechanism (*BL* column). In particular, both `gnulib` and `libbsd` have tests that make assumptions about the absolute number of file descriptors, and `gnulib` has several tests that call functions which alter the execution environment in a manner that affects other threads in the same process (e.g., calling `setrlimit` or changing the working directory). Our globals dependency analysis and blacklisting mechanism allow us to safely parallelize these test suites despite such issues. Since most globals dependencies originate in core project code itself, they are harder to remove for the purpose of parallel test execution as test suite modifications are insufficient in this case.

> *File dependencies occur in few projects and exclusively originate in test code, leading to flaky test behavior when not accounted for in parallel test execution. Globals dependencies are more common and frequently originate in the project itself.*

4.5.4 RQ 3: Achieved Speed-Ups

To assess the achievable speedups from concurrent test executions, we analyze how test suite execution times develop with increasing degrees of parallelism across the different execution modes.

As we found in our study of the Debian software repository (cf. Section 4.3.3) that some projects benefit from parallel `make`-based execution, we start by analyzing execution times obtained with `make` as our baseline. Figure 4.8 on the facing page, illustrates the observed speedups using the □ marker (y-axis, different scales) compared to *sequential* `make` execution (cf. Table 4.2 on page 106 for absolute sequential `make` execution times) for each project across increasing parallelism degrees (x-axis). We observe that 3 projects (`libbsd`, `libgetdata`, `librabbitmq`)

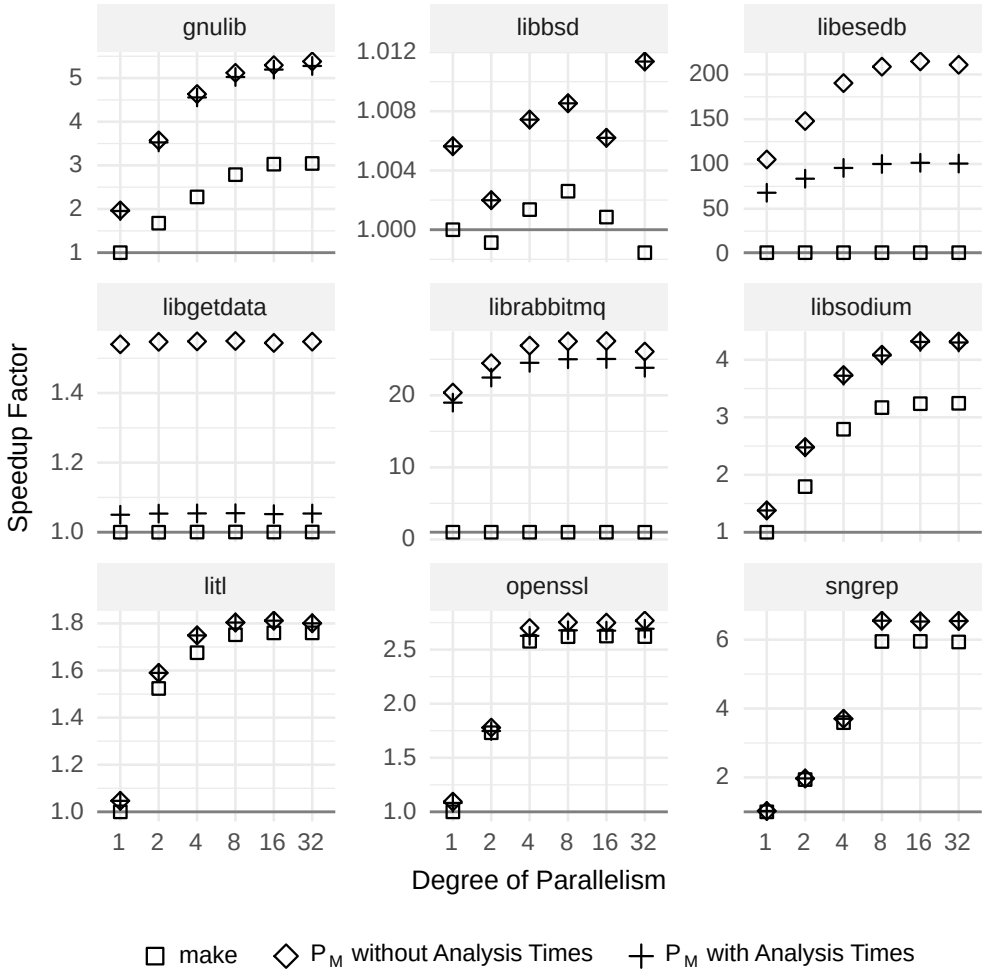


Figure 4.8: Parallel `make` and P_M speedups relative to sequential `make`-based execution. For P_M with analysis times, the file dependency analysis runtime was added to the test execution time.

do not show meaningful speedups with increasing parallelism for `make`, whereas the other 6 show speedups ranging from 1.02× to 5.95× (`sngrep`). `libgetdata` does not benefit from parallel `make` as sequential execution is hard-coded in its Makefile to respect its file dependencies. If we compare speedups achieved with our P_M mode (◇ in Figure 4.8), being conceptually closest to `make` (but respecting file dependencies), to `make` speedups, we see that our P_M mode consistently outperforms `make` with speedups over sequential `make` of 214× for the extreme case of `libesedb`, having extremely short tests (similarly to `librabbitmq`). Leaving out these extreme cases, we still see speedups over sequential `make` ranging from 1.01× to 6.55× (`sngrep`). The maximum relative speedup between parallel `make` and P_M was seen for `gnulib` with 2.13×. Remarkably, even sequential P_M execution is

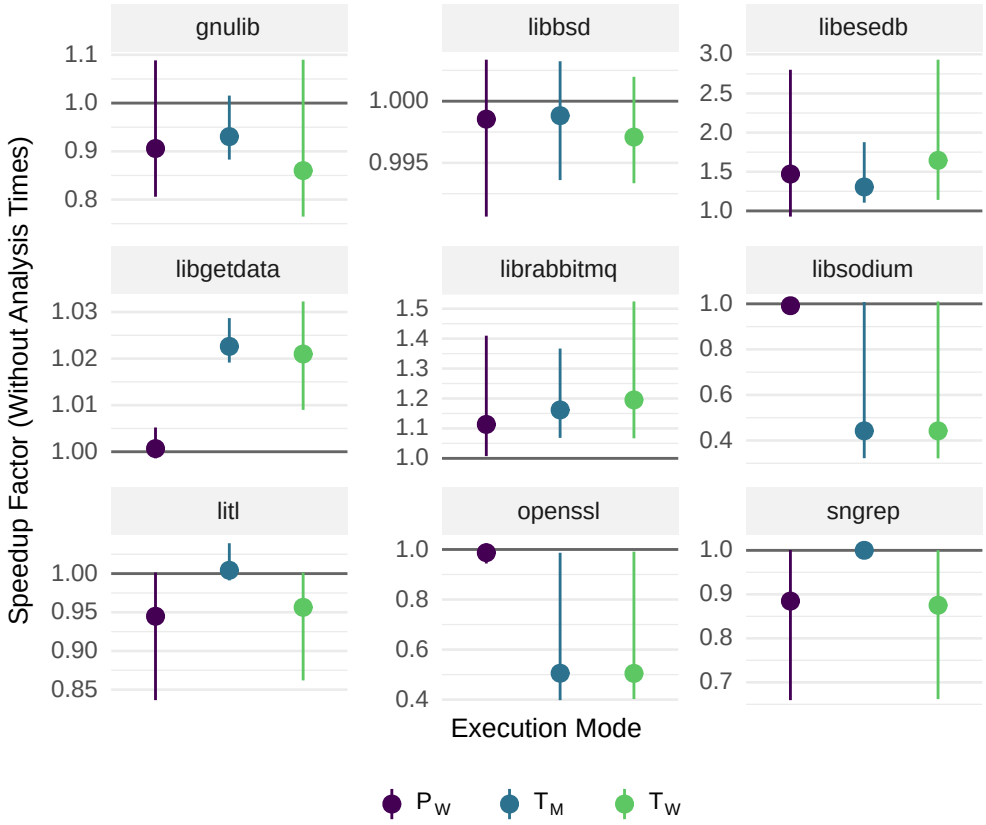


Figure 4.9: Geometric mean speedups relative to P_M at different degrees of parallelism, excluding analysis time overheads. Lines indicate minimum/maximum speedups. Speedups are computed based exclusively on execution times without taking required analysis times into account.

faster than sequential `make` execution (cf. Table 4.2 on page 106), which shows that `make` imposes a non-negligible overhead, being over 11 s for `gnulib` and over 18 s for `libgetdata`.

Comparing P_M to T_M and T_W, we observe that only 3 projects consistently benefit from multi-threaded test execution, which is illustrated in Figure 4.9 where the achieved speedups over P_M at respective degrees of parallelism for T_M, T_W, and P_W are shown in the upper part (geometric mean and min/max). `libesedb` and `librabbitmq` achieve a best case speedup of 1.9× for T_M and 2.9× for T_W, corresponding to less than 7 ms, whereas `libgetdata` achieves a minor speedup of up to 1.03× for both T_M and T_W, corresponding to 950 ms. We attribute the better multi-threaded performance for `libesedb` and `librabbitmq` to their extremely short tests (<1ms) where process creation overhead outweigh actual test execution. Similarly for `libgetdata`, we see the reason for the better T_M performance in the high number of short tests where over 95 % of tests are shorter than 5 ms. `openssl`

and `libbsd`, on the other hand, never benefit from T_M or T_W. All but the above 3 projects tend to perform worse in T_M/T_W than in P_M with a mean speedup of 1 or less with the extreme of `libsodium` with 0.4×.

To underpin our observation that multi-threading is not worthwhile compared to P_M, we perform a one-sided Wilcoxon signed-rank test with the null hypothesis that there is no execution time difference between P_M and T_M in the median and the alternative hypothesis that the median difference between P_M and T_M is positive. We perform the test for each project separately, pair the data points according to the parallelism degree, and use a significance level of $\alpha = 0.05$. For brevity, we omit the exact statistics and p-values; however, we were only able to reject the null hypothesis for the above mentioned 3 projects with p-values < 0.05 that showed geometric means speedups larger than 1. Hence, we cannot find statistically significant evidence that thread-based execution performs better than processes for the majority of studied projects.

Worker-based execution in P_W performs similar to T_W with the exception of `libsodium` and `openssl` where P_W, with a geometric mean speedup close to 1, performs better than T_W. However, worker-based execution perform sometimes slightly worse compared to other modes as tests have to be assigned to workers for serial execution without prior knowledge of individual test case durations, which can lead to suboptimal performance if multiple long running tests are assigned to the same worker. This effect can be observed for `gnulib`, `litl`, and `sngrep` where worker-based modes show slightly lower geometric mean speedups.

Two of the studied projects, `libbsd` and `libgetdata`, have comparatively long test suite execution times (cf. Table 4.1 on page 104) without a clear performance benefit of parallel execution. For `libbsd` a long running test case (`arc4random`) is the reason. For `libgetdata` file dependencies between virtually all test cases are the reason. To investigate the performance impact of such implementation decisions, we created variants where the long running test case of `libbsd` is restructured into 4 C functions that our analysis and test harness can recognize as test cases and the file dependencies in `libgetdata` have been removed by introducing unique filenames using a simple `sed` invocation. These very simple changes enable parallel execution in P_M mode with maximum speedups over `make` of up to 2.34× or 32.2 s for `libbsd` and 5.3× or 42.4 s for `libgetdata`.

Using P_M, we achieve parallel speedups of more than 2× over parallel and more than 6× over sequential `make`. Even sequentially, P_M consistently outperforms `make`, indicating that the use of a dedicated tool is preferable over `make`. Multi-threaded parallel execution is advantageous in only few cases with limited benefits.

4.5.5 RQ 4: Analysis Runtime Overhead and Amortization

To assess the run time overhead of our analyses, we run them on each project and measure the execution times. In the following, we consider the mean values of 30 repeated measurements for each project, which we report in the *Analysis Time* columns of Table 4.2 on page 106. Both our analyses finish in less than 1 s in all cases except for `libgetdata`, where our file analysis needs almost 16 s and our globals analysis 1.1 s to complete. This effect results from the high number of file dependencies and test cases in `libgetdata` (cf. Table 4.1 on page 104 and Table 4.2 on page 106). Reducing the number of file dependencies, as we did for the modified `libgetdata` variant discussed in Section 4.5.4, the file analysis time is reduced considerably by 16×. Overall, our file pass completed in less than 10 ms for 6 out of the remaining 8 cases and needed less than 87 ms for the other two. Except for `libgetdata`, our globals pass completed in less than 560 ms in all cases with `gnulib` taking the longest due to its large codebase and number of test cases. In all remaining 7 cases, our globals pass finishes in under 81 ms and in 5 cases in less than 10 ms.

To put the analysis run times into perspective, we relate them to the parallel test execution speedups that we achieve over `make`. We add the required analysis times for each project to the test execution time for our approach. As shown in Figure 4.8 on page 109 using the + marker, when adding the time required for file dependency analysis, P_M (x-axis) still outperforms `make`-based test execution (□ marker) for all projects across all degrees of parallelism. In the extreme case of `libesedb` the speedup is still up to 101× and for `sngrep` 6.55× over `make`. Looking at absolute time savings of P_M compared to `make` at respective parallelism degrees, we observed the best case for `gnulib` with 11.5 s saving. For our modified version of `libgetdata`, we saved up to 41.4 s. Overall, we observed savings between 15 ms and 1500 ms for 7 projects and savings above 2.5 s for the remaining 2 (excluding our two modified variants).

To assess the impact of the globals analysis time on the viability of the three modes that require it (T_M, T_W, P_W), we add file and globals analysis times to the test execution time for these modes and compute the resulting speedup relative to P_M with added file dependency analysis time. As shown in Figure 4.10 on the next page, this results in a best case speedup of just 1.15× over P_M for `librabbitmq` in T_W. However, no project exhibits a mean speedup significantly over 1.0× in any of T_M, T_W or P_W. For `libsodium` and `openssl`, using either of the thread-based modes T_M or T_W effectively halves the performance when taking the additional analysis time into account.

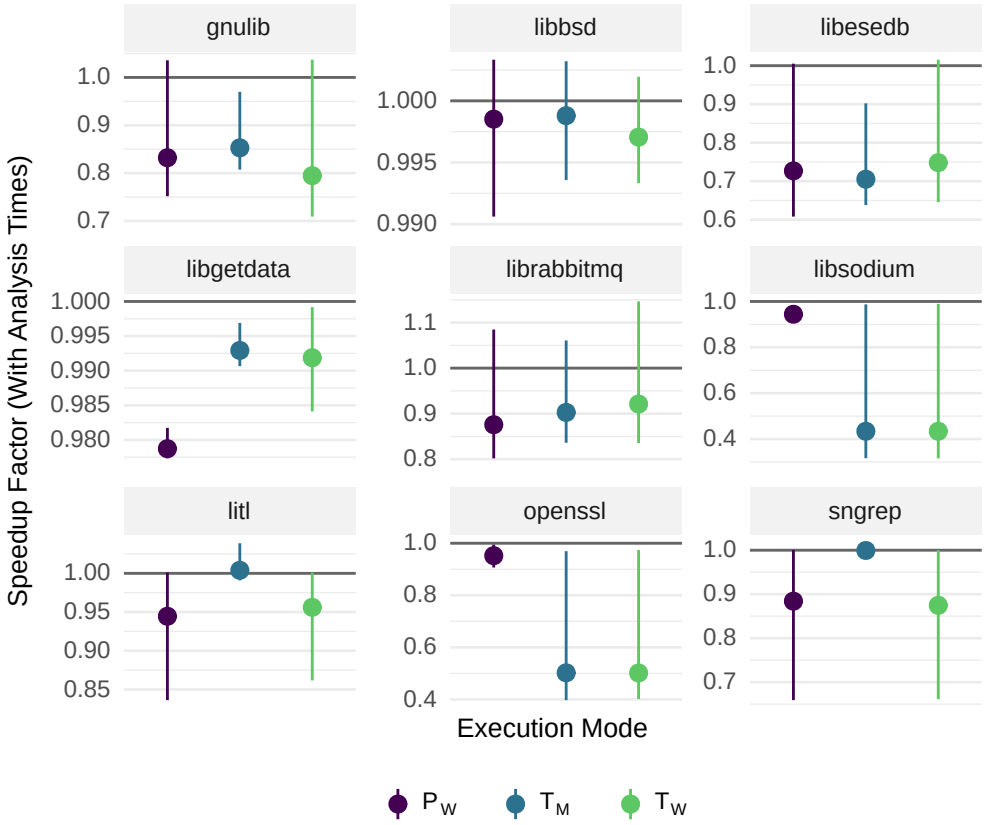


Figure 4.10: Geometric mean speedups relative to P_M at different degrees of parallelism with analysis time overheads included. Lines indicate minimum/maximum speedups. Speedups are computed with P_M times including file dependency analysis overhead as baseline and both file and globals analysis overhead included for P_W, T_M, and T_W.

> *The observed file analysis overheads are low enough to pay off for parallel test execution with P_M in all cases. The performance advantages of T_M, T_W, and P_W execution are not sufficient to justify the increased overhead for the globals analysis.*

4.5.6 Threats To Validity

Our analyses and conclusions depend on the selection of software projects and may not generalize to other software. We performed all our experiments on one platform (hardware and software), which may bias our results towards that single platform. We use platform supplied means for our time measurements and depend on their precision and accuracy.

4.6 Discussion & Lessons Learned

As we observed in our experiments, relying on `make` for test suite execution requires longer sequential execution times and achieves lower parallel speedups compared to our test harness. `libesedb` is an extreme example for this effect where `make` requires 2 orders of magnitude more execution time than P_M. `make`'s overhead can be saved by using tools that are tailored to test suite orchestration rather than a generic build automation tool like `make`. Hence, we recommend using specialized tools for test suite management. Such specialized tools should support the parallel execution of tests, as we observe parallel speedups with P_M in 7 out of 9 cases.

The observed performance of the multi-thread parallelization strategies was similar to the multi-process strategies. We expected to see both larger and more consistent differences in the execution times for P_M and T_M as both strategies spawn a new execution entity for each test, but thread creation is commonly considered a lighter operation than process forking. The 3 cases where we could observe a consistent performance advantage of multi-threading were those (1) that had very short test run times where the creation/cleanup of the execution entity dominates the overall execution time or (2) where a highly sequential execution was enforced in all modes (e.g., due to file conflicts) and the speedups achieved through parallelism could not compensate for the creation overhead of execution entities. As the analysis overhead required for multi-threaded execution eats up the small time savings these modes offer, we recommend P_M as the default choice for parallelization. The same considerations apply for the execution with a worker model (P_W, T_W) as we could not observe a clear performance benefit esp. when analysis overhead is taken into account.

For choosing a suitable parallel execution mode, the nature of the tests must be considered. Tests that persistently change their process environment without cleanup, e.g., changing working directories or changing environment variables, cannot safely execute in the same process. As tests are often designed with the implicit assumption that they execute in their own process, cleanup code is commonly omitted. Such tests are inherently unsuited for multi-threaded or worker-based execution and they need to be removed for modes other than P_M or cleanup code needs to be added, if possible. An extreme case, for which a cleanup is usually not possible, are tests that destroy their process, e.g., by explicitly aborting process execution, sending process signals, or causing segmentation faults. We opted to exclude such tests in our evaluation which is the reason for the reduction of test cases we report.

The achievable execution speedups depend on the parallelization potential of the test suite. The more test cases there are, the fewer dependencies they have, and the more similar the individual test case execution times are, the higher are the achievable speedups. Ideally, test suites would be designed with these goals in

mind. However, our study of the Debian repository and our evaluation indicate that only a fraction of C projects ship with test suites that already benefit from parallel execution. Hence, a migration path to parallel test suites is desirable to tap into the full potential of modern hardware for testing. Our approach offers such a migration path as we demonstrated in our evaluation that existing test suites can be converted with acceptable effort to benefit from parallel execution. We furthermore demonstrated (for `libgetdata`) that by mechanically removing file dependencies identified by our analysis, the achievable speedups can be increased considerably. The locations of conflicting globals and files we found suggest that existing test suites have further parallelization potential as a non-negligible number of dependencies originate in test code (cf. Table 4.2 on page 106).

The execution time savings we observed in our evaluation range from the order of tens of milliseconds to tens of seconds. These seem to be moderate savings in absolute numbers. However, when scaling to larger test suites or when conducting analyses on the ecosystem scale, these savings quickly accumulate to massive execution time savings. For instance, for conducting our experiments with `gnulib` in this chapter, we executed its test suite 30 times for each of the 6 degrees of parallelism. The total execution time for these experiments was about 35 min when executed with `make`, and only about 19 min when executed with P_M, which is almost a reduction by half.

4.7 Conclusion

In our study of the Debian "Buster" software repository, we found that C is the predominant language (28.6 % of total SLOC) and that only a fraction of C projects benefit from trivial parallel test execution using `make`. We showed that our approach of static dependency analysis with multi-thread and multi-process execution strategies is applicable to real world software in a study of nine software projects. We identified file dependencies in three and globals dependencies in five projects. All file dependencies originated in test code but most globals dependencies originated in the project code itself, suggesting that file dependencies can be removed by test suite modifications whereas globals dependencies cannot. Moreover, we can efficiently execute tests in parallel, even in the presence of such dependencies using our static analyses and test harness. We achieved test execution speedups over `make` of up to 210× in extreme cases and 2.1× in other cases with our multi-process strategy P_M. P_M outperforms `make` even in the sequential case, indicating that the use of a dedicated test orchestration tool is preferable over `make`. Multi-thread strategies did not show a consistent performance benefit for most projects we studied and offer no advantage when accounting for analysis time.

5 Summary and Conclusion

Computing systems have become ubiquitous in our daily lives. They take on surprising shapes and sizes, from small embedded systems to large scale servers, perform a multitude of tasks, and continue to be applied for new and innovative purposes. We are often hardly conscious of their presence, their full capabilities, and their inner workings. We as a society strive for a fully connected and automated world to simplify our lives and increase comfort. This vision is driven by an ever growing ecosystem of software that enables the increasingly complex functions and applications we demand. To cope with the demand for smarter and more complex applications, software and systems developers are relying on the re-use of old components as well as on off-the-shelf components. These components need to be correctly composed and orchestrated together with newly developed components. In this scenario, the question arises whether we can trust these complex systems and software stacks, we have become so reliant on, to operate correctly and perform the expected tasks. Software is developed by human beings under constraints such as development cost budgets. Hence, software must be expected to contain defects or bugs, with a wide range of consequences when triggered that can be particularly severe in the case of safety-critical systems.

Various approaches are commonly combined to improve the dependability of our software. For instance, isolation mechanisms are employed to limit the effects a triggered defect can have. Dynamic correctness testing is commonly used to establish a certain level of confidence that a software is performing the intended tasks according to its specification. An alternative technique is SFI testing that is used to assess and improve the robustness of a system by exposing it to anomalous conditions or by inserting artificial software defects while observing and diagnosing the resulting behavior. However, due to the increase in software complexity, such testing approaches suffer from an exploding number of tests which leads to long latencies when software is assessed.

This thesis investigated these issues from three points of view: First, from a fault isolation perspective with the goal of balancing the performance overheads with amount of isolated code. Second, from a software fault injection perspective with the goal to assess if and how the increasing computational power of modern hardware and other techniques can be leveraged to mitigate the increasing number of SFI tests. And third, from a software testing perspective with the similar goal to assess if parallel hardware can already be used by real-world test suites and to

provide strategies to further increase the utility of such hardware. In summary, this thesis investigated the following research questions along with the resulting contributions.

Research Question 1 (RQ 1): Can runtime profiling be leveraged for the partitioning of in-kernel software components to increase code isolation while balancing performance overhead?

The most widely used OSs, with Linux being a prominent example, employ a monolithic kernel design where all kernel internal components execute together in high-privilege mode without any isolation between components. This design, while having good performance properties, potentially endangers the whole system as defects in any in-kernel component can affect the whole kernel. To preserve backwards compatibility to the large existing codebase and improve dependability, approaches that "microkernelize" existing kernel components by moving fractions of their code to user mode have been proposed. Unfortunately, these conversion approaches do not take dynamic properties, such as code invocation frequencies, of the targeted components into account. This information, however, is vital to generate component partitionings that are favorable in terms of both code isolation and achievable performance.

Contribution 1 (C 1): Runtime profiling based approach to tailor partitioning to performance needs

In Chapter 2, we have presented a largely automated approach that provides the needed guidance to generate partitionings of existing in-kernel software components into user and kernel partitions that respect a user configured balance between the amount of code that is isolated in user mode and the expected performance overhead this isolation entails. We re-use an already existing framework that provides us with the mechanisms for such hybrid split mode components. Our approach combines static code analysis with dynamic runtime profiling. For profiling, we generate an instrumented variant of the targeted component that is then exercised. We model the user/kernel mode partitioning problem as a binary ILP problem and rely on a linear solver to find an optimal solution with respect to the configured balance factor. In general, the more code is relocated to user mode, the better the isolation properties but the worse the expected performance. We implemented our approach for the Linux kernel and validated its applicability by profiling and partitioning two device drivers and a file system. In our evaluation, we generated a large spectrum of partitionings for each targeted component with varying balance factors for the expected isolation performance trade-off. We synthesized hybrid split mode version for all partitionings and measured the achieved performance. This demonstrates that our approach is adaptable to use specified requirements

as the overhead one is prepared to pay for improved isolation is freely choosable. Finally, we used software fault injection experiments to demonstrate the impact of defects depending on their location, i.e., whether they are located in the user or the kernel partition, thereby showing dependability benefits of having larger user partition sizes as the kernel indeed crashed if the injected defects were activated in kernel mode.

Research Question 2 (RQ 2): How can parallel hardware be exploited to increase the efficiency of software fault injections?

With increasing software complexity comes the need to execute vast numbers of SFI experiments for comprehensive robustness assessments. Exploiting the capabilities of parallel hardware is an obvious mitigation approach for the increased experiment campaign durations. However, SFI test results obtained from parallel experiments must be as accurate as sequential executions to be useful, which is often automatically assumed but may not necessarily be the case in practice. If SFI test results are not accurate when obtained from parallel executions, they should not be used for dependability assessments.

Contribution 2 (C 2): A framework for increasing the throughput of SFI tests by parallel execution and avoiding redundant work

In Chapter 3, we presented our study of PAIN experiments and introduced FastFI, leveraging the gained insights. We conducted our PAIN, i.e., PAralell fault INjection, experiments on the Android OS using the MMC device driver. As the attempt to parallelize SFI experiments relies on the assumption that the parallel execution of experiments does not impact the results, our goal was to investigate if the parallel execution of SFI experiments does indeed increase the throughput and if this supposed increase in throughput comes at a cost in terms of degraded result accuracy. Consequently, we assessed the supposed trade-off between achievable increase in experiment throughput and the accuracy of obtained results. We indeed were able to find several causes that can lead to a significant deviation of parallel SFI results compared to results obtained from sequential experiments and provide guidance on how to avoid such issues. We found that the degree of employed parallelism and the choice of timeout thresholds for timeout-based failure detectors must be carefully chosen to prevent resource contention and the timing of events from distorting result distributions. If the machine that hosts the experiments is overloaded or timeout thresholds are set too low, results start deviating.

We then introduced FastFI, which is our SFI framework for applications above the OS layer that leverages the insights from the PAIN study. FastFI accelerates SFI testing on multiple levels by combining different techniques and strategies. It relies on lightweight process-based isolation, avoiding unnecessary overheads

from heavier isolation mechanisms, such as VMs; it avoids the execution of a large fraction of "dead" mutants, i.e., faulty component versions whose fault cannot be reached during execution; it avoids the re-execution of redundant common prefixes across faulty versions; it employs parallel execution of faulty versions; and it significantly accelerates the compilation process of faulty versions as all faulty versions are integrated into a common executable. Our evaluation demonstrated that FastFI is applicable to real applications and that results remain accurate if, similar to our PAIN experiments, timeout thresholds and degree of parallelism are sensibly chosen.

Research Question 3 (RQ 3): What is the state of parallel testing for C software and can it be improved to reduce test suite execution latencies?

Testing is one of the most time consuming activities in software development, of which the dynamic execution of test suites is an important part to assess the correct function of the software under test. Relying on parallel hardware to reduce the latency of test suite executions is an obvious strategy to accelerate the overall testing process. However, there is a large body of testing code that was not originally designed for parallel or concurrent execution. If individual test cases interfere with each other when executed in parallel, the results obtained from such parallel executions may deviate significantly from those obtained from sequential execution, rendering the test results useless. Such interferences may be prevented by execution each test in an isolated environment, such as inside a VM, but isolation imposes performance overheads that diminish the benefit of the parallel execution. Therefore, it is preferable to rely on lightweight isolation mechanisms and orchestrate individual tests in a safe and efficient manner to prevent said issues.

Contribution 3 (C 3): An assessment of real world C software test suites and an approach for safe concurrent execution of existing tests

In Chapter 4, we investigated the potential for parallel test execution for C software, being an important building block across the software stack. We presented an analysis of the main software package repository of the Debian Buster OS, which is a widely used Linux-based OS distribution. In this analysis, we inspected the source code of a large fraction of software that is by default included in the Debian OS. Our results show that the C language is the predominant language both in terms of software packages that use C as their main language as well as by the total number of source lines of code contained in the entire repository. The analysis furthermore showed that there is no test framework that dominates C software packages, in fact we could not identify the use of any such framework for most packages. This is in strong contrast to Java-based packages, which commonly use

the JUnit framework. Finally, our data showed that few test suite implementations can benefit from out-of-the-box concurrent execution. We therefore continued with the development of an automated static analysis for existing C test suites that identifies test case interdependencies on files and shared global data and can be integrated into the software build process. We designed a new test harness to use the dependency information obtained from our static analysis for the safe parallel execution of independent tests and to explore the trade-off between analysis overheads and execution latencies for different parallelization alternatives using processes and threads. We demonstrated the utility of our approach by applying it to nine projects from the Debian Buster software repository, analyzing their existing tests and executing them in parallel orchestrated by our test harness. The observed measurements indicate that C test suites can benefit from parallel execution, but thread-based execution does not perform significantly better than processes, in particular when analysis overheads are considered, and that our test harness outperforms generic automation tools like make.

With our growing dependence on increasingly complex software systems and smart devices, it is of utmost importance that we assess and ensure their dependability. To that end, numerous techniques have been developed over the past decades for improving software dependability. These techniques must be efficiently applicable to increasingly complex systems. We have considered fault isolation, software fault injection, and software testing techniques, and developed approaches to improve efficiency for each of them. Such approaches increase the practicality of applying dependability-improving techniques to large, complex software systems that may otherwise be out of reach, thereby contributing to software dependability.

List of Figures

List of Tables

Bibliography

[ABL05] J.H. Andrews, L.C. Briand, and Y. Labiche. "Is mutation an appropriate tool for testing experiments?" In: *Proceedings of the 27th international conference on Software engineering*. ICSE'05. 2005, pp. 402–411. DOI: 10.1145/1062455.1062530.

[Adv19] Advanced Micro Devices, Inc. *AMD Processor Specifications*. 2019. URL: https://www.amd.com/en/products/specifications/processors (visited on 02/10/2019).

[Aid+01] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. "GOOFI: Generic Object-Oriented Fault Injection Tool". In: *2001 International Conference on Dependable Systems and Networks*. 2001, pp. 83–88. DOI: 10.1109/DSN.2001.941394.

[Arl+02] J. Arlat, J.C. Fabre, M. Rodríguez, and F. Salles. "Dependability of COTS Microkernel-Based Systems". In: *IEEE Transactions on Computers* 51.2 (2002), pp. 138–163. DOI: 10.1109/12.980005.

[Avi+04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[AY18] Sonny Ali and Zia Yusuf. *Mapping the Smart-Home Market*. 2018. URL: https://www.bcg.com/publications/2018/mapping-smart-home-market.aspx (visited on 02/10/2019).

[Ban+10] Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato. "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology". In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 2010, pp. 631–636. DOI: 10.1109/CCGRID.2010.72.

[Bar18] Barr Group. *Embedded Systems Safety & Security 2018 Survey*. Survey. 2018.

[Bav+] G. Bavota, A De Lucia, A Marcus, and R. Oliveto. "Software Re-Modularization Based on Structural and Semantic Metrics". In: *Proc. of the 17th Working Conference on Reverse Engineering*. WCRE '10. DOI: 10.1109/WCRE.2010.29.

[BC12] Radu Banabic and George Candea. "Fast black-box testing of system recovery code". In: *Proceedings of the 7th ACM european conference on Computer Systems*. EuroSys'12. 2012, pp. 281–294. DOI: 10.1145/2168836.2168865.

[Bei03] Boris Beizer. *Software Testing Techniques*. 2nd ed. Dreamtech Press, 2003.

[Bel+15] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. "Efficient dependency detection for safe Java test acceleration". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE'15. New York, New York, USA: ACM Press, 2015, pp. 770–781. DOI: 10.1145/2786805.2786823.

[Bel17] Fabrice Bellard. *QEMU*. 2017. URL: https://www.qemu.org.

[BH95] Yoav Benjamini and Yosef Hochberg. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 57.1 (1995), pp. 289–300.

[Bie11] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[Bis+13] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère. "Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects". In: *2013 IEEE 37th Annual Computer Software and Applications Conference*. July 2013, pp. 303–312. DOI: 10.1109/COMPSAC.2013.55.

[Bit+08] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-privilege Compartments". In: *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. USENIX Association, 2008.

[BK14] Jonathan Bell and Gail Kaiser. "Unit Test Virtualization with VMVM". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 550–561. DOI: 10.1145/2568225.2568248.

[Bov+11] Antonio Bovenzi, Marcello Cinque, Domenico Cotroneo, Roberto Natella, and Gabriella Carrozza. "OS-Level Hang Detection in Complex Software Systems". In: *Int. J. Crit. Comput.-Based Syst.* 2.3/4 (Sept. 2011), pp. 352–377.

[BS04] David Brumley and Dawn Song. "Privtrans: Automatically Partitioning Programs for Privilege Separation". In: *Proc. of the 13th USENIX Security Symposium*. USENIX Security '04. 2004.

[Bud+80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs". In: *Proc. POPL*. 1980, pp. 220–233.

[But+09] Shakeel Butt, Vinod Ganapathy, Michael M Swift, and Chih-Cheng Chang. "Protecting Commodity Operating System Kernels from Vulnerable Device Drivers". In: *Proc. of the Annual Computer Security Applications Conference*. ACSAC '09. 2009. DOI: 10.1109/ACSAC.2009.35.

[Cas+09] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. "Fast Byte-granularity Software Fault Isolation". In: *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. ACM, 2009. DOI: 10.1145/1629575.1629581.

[CB89] R. Chillarege and N. Bowen. "Understanding large system failures-a fault injection experiment". In: *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1989, pp. 356–363. DOI: 10.1109/FTCS.1989.105592.

[CBZ10] George Candea, Stefan Bucur, and Cristian Zamfir. "Automated Software Testing as a Service". In: *Proceedings of the 1st ACM symposium on Cloud computing*. SOCC'10. 2010, pp. 155–160. DOI: 10.1145/1807128.1807153.

[CC96] J. Christmansson and R. Chillarege. "Generation of an error set that emulates software faults based on field data". In: *Proceedings of Annual Symposium on Fault Tolerant Computing*. June 1996, pp. 304–313. DOI: 10.1109/FTCS.1996.534615.

[Cha+04] Ramesh Chandra, Ryan M Lefever, Kaustubh R Joshi, Michel Cukier, and William H Sanders. "A global-state-triggered fault injector for distributed system evaluation". In: *IEEE Transactions on Parallel and Distributed Systems* 15.7 (2004), pp. 593–605. DOI: 10.1109/TPDS.2004.14.

[Cha09] Robert N. Charette. "This Car Runs on Code". In: *IEEE Spectrum* 46 (3 2009).

[Cho+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. "An Empirical Study of Operating Systems Errors". In: *Proc. of the eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. 2001, pp. 73–88. DOI: 10.1145/502034.502042.

[Chu+11] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. "CloneCloud: Elastic Execution Between Mobile Device and Cloud". In: *Proc. of the Sixth Conference on Computer Systems*. EuroSys '11. ACM, 2011. DOI: 10.1145/1966445.1966473.

[Cio+10] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. "Cloud9: A Software Testing Service". In: *SIGOPS Oper. Syst. Rev.* 43.4 (Jan. 2010), pp. 5–10.

[Cis18] Cisco. *Cisco Visual Networking Index: Forecast and Trends, 2017–2022*. White Paper, ID 1543280537836565. Nov. 26, 2018. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html (visited on 02/10/2019).

[CK16] Jonathan Corbet and Greg Kroah-Hartman. *Linux Kernel Development: How Fast It is Going, Who is Doing It, What They Are Doing and Who is Sponsoring the Work*. 25th Anniversary. The Linux Foundation, Aug. 2016.

[CMd17] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. "Test Suite Parallelization in Open-source Projects: A Study on Its Usage and Impact". In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 838–848.

[CMS98] João Carreira, Henrique Madeira, and João Gabriel Silva. "Xception: A technique for the experimental evaluation of dependability in modern computers". In: *IEEE Transactions on Software Engineering* 24.2 (1998), pp. 125–136. DOI: 10.1109/32.666826.

[CN13] D. Cotroneo and R. Natella. "Fault Injection for Software Certification". In: *IEEE Security Privacy* 11.4 (2013), pp. 38–45. DOI: 10.1109/MSP.2013.54.

[CNR09] D. Cotroneo, R. Natella, and S. Russo. "Assessment and Improvement of Hang Detection in the Linux Operating System". In: *2009 28th IEEE International Symposium on Reliable Distributed Systems*. Sept. 2009, pp. 288–294. DOI: 10.1109/SRDS.2009.26.

[Col+11] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. "Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor". In: *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. ACM, 2011. DOI: 10.1145/2043556.2043575.

[Cop+17] Nicolas Coppik, Oliver Schwahn, Stefan Winter, and Neeraj Suri. "TrEKer: Tracing Error Propagation in Operating System Kernels". In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 377–387. DOI: 10.1109/ASE.2017.8115650.

[Cor08] Jonathan Corbet. *The big kernel lock strikes again [LWN.net]*. May 13, 2008. URL: https://lwn.net/Articles/281938/.

[Cot+13a] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S Trivedi. "Fault Triggers in Open-Source Software: An Experience Report". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 2013, pp. 178–187. DOI: 10.1109/ISSRE.2013.6698917.

[Cot+13b] Domenico Cotroneo, Roberto Natella, Stefano Russo, and Fabio Scippacercola. "State-Driven Testing of Distributed Systems". In: *Proc. OPODIS'13*. 2013, pp. 114–128.

[CSS19] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. "MemFuzz: Using Memory Accesses to Guide Fuzzing". In: *12th IEEE International Conference on Software Testing, Verification and Validation*. ICST 2019. Xi'an, China, Apr. 2019. [accepted].

[CV95] A. Cimitile and G. Visaggio. "Software Salvaging and the Call Dominance Tree". In: *Journal of Systems and Software* 28.2 (Feb. 1995), pp. 117–127. DOI: 10.1016/0164-1212(94)00049-S.

[Dan18] Al Danial. *cloc*. Jan. 2018. URL: https://github.com/AlDanial/cloc (visited on 01/26/2018).

[Deb17] Debian Project. *Debian Buster Source Package Index*. en. July 2017. URL: http://ftp.debian.org/debian/dists/buster/main/source/Sources.gz (visited on 07/18/2017).

[Deb18] Debian Wiki Team. *Debian Derivatives Census*. 2018. URL: https://wiki.debian.org/Derivatives/Census (visited on 01/31/2019).

[DH12] Björn Döbel and Hermann Härtig. "Who Watches the Watchmen? - Protecting Operating System Reliability Mechanisms". In: *Proc. of the Eighth USENIX Conference on Hot Topics in System Dependability*. HotDep'12. USENIX Association, 2012.

[Di +12] Domenico Di Leo, Fatemeh Ayatolahi, Behrooz Sangchoolie, Johan Karlsson, and Roger Johansson. "On the Impact of Hardware Faults–An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions". In: *Proc. of the 31st International Conference on Computer Safety, Reliability, and Security*. SAFECOMP'12. 2012, pp. 198–209. DOI: 10.1007/978-3-642-33678-2_17.

[DK99] Arie van Deursen and Tobias Kuipers. "Identifying Objects Using Cluster and Concept Analysis". In: *Proc. of the 21st International Conference on Software Engineering*. ICSE '99. ACM, 1999. DOI: 10.1145/302405.302629.

[DM] DEEDS/TUD and Mobilab/UniNa. *PAIN Software Framework*. URL: https://github.com/DEEDS-TUD/PAIN.git.

[DM03] Joao Duraes and Henrique Madeira. "Multidimensional characterization of the impact of faulty drivers on the operating systems behavior". In: *IEICE Transactions on Information and Systems* 86.12 (2003), pp. 2563–2570.

[DM06] J. A. Duraes and H. S. Madeira. "Emulation of Software faults: A Field Data Study and a Practical Approach". In: *IEEE Transactions on Software Engineering* 32.11 (2006), pp. 849–867. DOI: 10.1109/TSE.2006.113.

[DO91] R. A. DeMillo and A. J. Offutt. "Constraint-based automatic test data generation". In: *IEEE Transactions on Software Engineering* 17.9 (Sept. 1991), pp. 900–910.

[DR06] H. Do and G. Rothermel. "On the use of mutation faults in empirical assessments of test case prioritization techniques". In: *IEEE Transactions on Software Engineering* 32.9 (2006), pp. 733–752. DOI: 10.1109/TSE.2006.92.

[Dua+06] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. "GridUnit: Software Testing on the Grid". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. New York, NY, USA: ACM, 2006, pp. 779–782. DOI: 10.1145/1134285.1134410.

[DVM04] Joao Duraes, Marco Vieira, and Henrique Madeira. "Dependability Benchmarking of Web-Servers". In: *Computer Safety, Reliability, and Security*. Vol. 3219. Lecture Notes in Computer Science. 2004, pp. 297–310.

[ED12] Michael Engel and Björn Döbel. "The Reliable Computing Base: A Paradigm for Software-Based Reliability". In: *Workshop on Software-Based Methods for Robust Embedded Systems*. 2012.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Fre] Free Software Foundation. *GLPK (GNU Linear Programming Kit)*. URL: https://www.gnu.org/software/glpk/ (visited on 08/01/2016).

[Gam+17] Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. "CUT: Automatic Unit Testing in the Cloud". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 364–367. DOI: 10.1145/3092703.3098222.

[Gan+08] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. "The Design and Implementation of Microdrivers". In: *Proc. of the 13th international conference on Architectural support for programming languages and operating systems*. ASPLOS XIII. 2008. DOI: http://doi.acm.org/10.1145/1346281.1346303.

[Gan05] Archana Ganapathi. *Why Does Windows Crash?* Tech. rep. CSD-05-1393. UC Berkeley, May 2005.

[GBZ18] A. Gambi, J. Bell, and A. Zeller. "Practical Test Dependency Detection". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2018, pp. 1–11. DOI: 10.1109/ICST.2018.00011.

[Gef+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. "The SawMill Multiserver Approach". In: *Proc. of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*. EW 9. ACM, 2000. DOI: 10.1145/566726.566751.

[GGP06] Archana Ganapathi, Viji Ganapathi, and David Patterson. "Windows XP Kernel Crash Analysis". In: *Proc. of the 20th Conference on Large Installation System Administration*. LISA '06. USENIX Association, 2006, pp. 12–22.

[GGZ17] A. Gambi, A. Gorla, and A. Zeller. "O!Snap: Cost-Efficient Testing in the Cloud". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017, pp. 454–459. DOI: 10.1109/ICST.2017.51.

[GJ10] Weigang Gong and Hans-Arno Jacobsen. *ACC: The AspeCt-oriented C Compiler*. 2010. URL: https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc (visited on 08/01/2016).

[Gooa] Google Inc. *Android*. URL: https://www.android.com.

[Goob] Google Inc. *Android Emulator*. URL: http://developer.android.com/tools/help/emulator.html.

[Gooc] Google Inc. *android Git repositories*. URL: https://android.googlesource.com/.

[GT07] M. Grottke and K.S. Trivedi. "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate". In: *IEEE Computer* 40.2 (2007), pp. 107–109.

[Gun+11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. "FATE and DESTINI: A Framework for Cloud Recovery Testing". In: *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. 2011, pp. 238–252.

[Gup+06] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. "Enforcing Performance Isolation Across Virtual Machines in Xen". In: *Proc. Middleware*. 2006, pp. 342–362.

[GVS17] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. "An Empirical Study of Activity, Popularity, Size, Testing, and Stability in Continuous Integration". In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 495–498. DOI: 10.1109/MSR.2017.38.

[Gyo+15] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. "Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 223–233. DOI: 10.1145/2771783.2771793.

[Han+10] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato. "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems". In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 2010, pp. 428–433. DOI: 10.1109/ICSTW.2010.59.

[Her+09] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. "Fault Isolation for Device Drivers". In: *Proc. of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '09. June 2009. DOI: 10.1109/DSN.2009.5270357.

[HHP02] Mark Harman, Robert M. Hierons, and Mark Proctor. "A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization". In: *Proc. of the Genetic and Evolutionary Computation Conference*. GECCO '02. Morgan Kaufmann Publishers Inc., 2002.

[HL13] Qun Huang and Patrick PC Lee. "An experimental study of cascading performance interference in a virtualized environment". In: *ACM SIGMETRICS Performance Evaluation Review* 40.4 (2013), pp. 43–52.

[Hoh+04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. "Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-machine Monitors". In: *Proc. of the 11th Workshop on ACM SIGOPS European Workshop*. EW 11. ACM, 2004. DOI: 10.1145/1133572.1133615.

[Hru+12] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. "Keep net working - on a dependable and fast networking stack". In: *Proc. of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '12. June 2012. DOI: 10.1109/DSN.2012.6263933.

[HTI97] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. "Fault Injection Techniques and Tools". In: *Computer* 30.4 (Apr. 1997), pp. 75–82.

[IDC19] IDC. *Smartphone Market Share – OS (2016 – 2018)*. 2019. URL: https://www.idc.com/promo/smartphone-market-share/os (visited on 02/01/2019).

[IEE18] IEEE and The Open Group. "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (Jan. 2018), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[INR18] INRIA. *Coccinelle Website*. 2018. URL: http://coccinelle.lip6.fr (visited on 01/31/2019).

[Int10] International Electrotechnical Commission. *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. 2010.

[Int11] International Organization for Standardization. *ISO 26262: Road Vehicles – Functional Safety*. 2011.

[Irr+13] Ivano Irrera, João Durães, Henrique Madeira, and Marco Vieira. "Assessing the Impact of Virtualization on the Generation of Failure Prediction Data". In: *2013 Sixth Latin-American Symposium on Dependable Computing*. 2013, pp. 92–97. DOI: 10.1109/LADC.2013.24.

[Jai+14] Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E. Porter. "Practical Techniques to Obviate Setuid-to-root Binaries". In: *Proc. of the Ninth European Conference on Computer Systems*. EuroSys '14. ACM, 2014. DOI: 10.1145/2592798.2592811.

[JGS11] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. "PREFAIL: A Programmable Tool for Multiple-failure Injection". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 2011, pp. 171–188. DOI: 10.1145/2048066.2048082.

[JH08] Yue Jia and M. Harman. "Constructing Subtle Faults Using Higher Order Mutation Testing". In: *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. Sept. 2008, pp. 249–258. DOI: 10.1109/SCAM.2008.36.

[JH09] Yue Jia and Mark Harman. "Higher Order Mutation Testing". In: *Information and Software Technology* 51.10 (2009), pp. 1379–1393.

[JH11] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678.

[Jo+10] Heeseung Jo, Hwanju Kim, Jae-Wan Jang, Joonwon Lee, and Seungryoul Maeng. "Transparent Fault Tolerance of Device Drivers for Virtual Machines". In: *IEEE Transactions on Computers* 59.11 (Nov. 2010), pp. 1466–1479. DOI: 10.1109/TC.2010.61.

[Kan09] Antti Kantee. "Rump File Systems: Kernel Code Reborn". In: *Proc. of the 2009 USENIX Annual Technical Conference*. USENIX'09. 2009.

[Kap01] Gregory M Kapfhammer. "Automatically and Transparently Distributing the Execution of Regression Test Suites". In: *Proc. of the 18th International Conference on Testing Computer Software*. 2001.

[KD00] P. Koopman and J. DeVale. "The Exception Handling Effectiveness of POSIX Operating Systems". In: *IEEE Transactions on Software Engineering* 26.9 (2000), pp. 837–848. DOI: 10.1109/32.877845.

[KDD08] Philip Koopman, Kobey DeVale, and John DeVale. "Interface robustness testing: Experience and lessons learned from the ballista project". In: *Dependability Benchmarking for Computer Systems* 72 (2008), p. 201.

[Ker] Gabriel Kerneis. *CIL (C Intermediate Language)*. URL: https://github.com/cil-project/cil (visited on 08/01/2016).

[KGT14] Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. "Evaluating Distortion in Fault Injection Experiments". In: *Proc. HASE'14*. 2014.

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". In: *ECOOP'97 – Object-Oriented Programming*. Ed. by Mehmet Aksit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 220–242. DOI: 10.1007/BFb0053381.

[Kil03] Douglas Kilpatrick. "Privman: A Library for Partitioning Applications". In: *Proc. of the FREENIX Track: 2003 USENIX Annual Technical Conference*. 2003.

[KIT93] W. I. Kao, R. K. Iyer, and D. Tang. "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults". In: *IEEE Transactions on Software Engineering* 19.11 (Nov. 1993), pp. 1105–1118.

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel". In: *Proc. of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. ACM, 2009. DOI: 10.1145/1629575.1629596.

[Kle+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014). DOI: 10.1145/2560537.

[Koc+13] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. "An Empirical Study of Adoption of Software Testing in Open Source Projects". In: *2013 13th International Conference on Quality Software*. July 2013, pp. 103–112. DOI: 10.1109/QSIC.2013.57.

[KS08] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.

[LA02] Timothy C. Lethbridge and Nicolas Anquetil. "Approaches to Clustering for Program Comprehension and Remodularization". In: *Advances in Software Engineering*. Ed. by Hakan Erdogmus and Oryal Tanir. Springer-Verlag New York, Inc., 2002, pp. 137–157.

[LA04] Chris Lattner and Vikram 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 '04. Palo Alto, California: IEEE Computer Society, 2004.

[Lan+14] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. "An Empirical Study of Injected versus Actual Interface Errors". In: *Proc. ISSTA*. 2014, pp. 397–408.

[Las05] Alexey Lastovetsky. "Parallel testing of distributed software". In: *Information and Software Technology* 47.10 (2005), pp. 657–662.

[LeV+04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines". In: *Proc. of the 6th Symposium on Operating Systems Design & Implementation*. OSDI'04. USENIX Association, 2004.

[Levo4] Nancy G. Leveson. "Role of Software in Spacecraft Accidents". In: *Journal of Spacecraft and Rockets* 41.4 (July 2004), pp. 564–575. DOI: 10.2514/1.11950.

[Li+14] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. "MiniBox: A Two-way Sandbox for x86 Native Code". In: *Proc. of the 2014 USENIX Annual Technical Conference*. USENIX ATC '14. USENIX Association, 2014.

[Lie] J. Liedtke. "On μ-Kernel Construction". In: *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. DOI: 10.1145/224056.224075.

[Lio+96] Jacques-Louis Lions et al. *Ariane 5 Flight 501 Failure*. 1996.

[Liu+15] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. "Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation". In: *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. ACM, 2015. DOI: 10.1145/2810103.2813690.

[Lon] Roy Longbottom. *Roy Longbottom's Android Benchmark Apps*. URL: http://www.roylongbottom.org.uk/android%20benchmarks.htm.

[Lov] Robert Love. *Sleeping in the interrupt handler*. URL: http://permalink.gmane.org/gmane.linux.kernel.kernelnewbies/1791 (visited on 02/01/2016).

[LT93] Nancy G. Leveson and Clark S. Turner. "An Investigation of the Therac-25 Accidents". In: *Computer* 26.7 (July 1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[LZE15] Wing Lam, Sai Zhang, and Michael D. Ernst. *When tests collide: Evaluating and coping with the impact of test dependence*. Tech. rep. University of Washington Department of Computer Science and Engineering, 2015.

[LZN04] Chung-Horng Lung, Marzia Zaman, and Amit Nandi. "Applications of clustering techniques to software partitioning, recovery and restructuring". In: *Journal of Systems and Software* 73.2 (2004), pp. 227–244. DOI: http://dx.doi.org/10.1016/S0164-1212(03)00234-6.

[Mah+12] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. "A whitebox approach for automated security testing of Android applications on the cloud". In: *Proc. of the 7th International Workshop on Automation of Software Test*. 2012, pp. 22–28. DOI: 10.5555/2663608.2663613.

[Mao+11] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. "Software Fault Isolation with API Integrity and Multi-principal Modules". In: *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. ACM, 2011. DOI: 10.1145/2043556.2043568.

[MB07] O. Maqbool and H.A Babri. "Hierarchical Clustering for Software Architecture Recovery". In: *IEEE Transactions on Software Engineering* 33.11 (Nov. 2007), pp. 759–780. DOI: 10.1109/TSE.2007.70732.

[Mis+07] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. "Parallel Test Generation and Execution with Korat". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 135–144. DOI: 10.1145/1287624.1287645.

[MM06] B.S. Mitchell and S. Mancoridis. "On the Automatic Modularization of Software Systems Using the Bunch Tool". In: *IEEE Transactions on Software Engineering* 32.3 (Mar. 2006), pp. 193–208. DOI: 10.1109/TSE.2006.31.

[MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. "Improving Xen Security Through Disaggregation". In: *Proc. of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '08. ACM, 2008. DOI: 10.1145/1346256.1346278.

[MN07] M. Mendonça and N. Neves. "Robustness Testing of the Windows DDK". In: *Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '07. 2007.

[MN10] Antti P. Miettinen and Jukka K. Nurminen. "Energy Efficiency of Mobile Clients in Cloud Computing". In: *Proc. of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. 2010.

[MSB12] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd ed. John Wiley & Sons Inc., 2012.

[MSW11] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. "Finding Bugs by Isolating Unit Tests". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 496–499. DOI: 10.1145/2025113.2025202.

[MWC10] Adrian Mettler, David Wagner, and Tyler Close. "Joe-E: A Security-Oriented Subset of Java". In: *Proc. of 17th Annual Network and Distributed System Security Symposium*. NDSS '10. 2010.

[Nat+13] R. Natella, D. Cotroneo, J.A. Durães, and H.S. Madeira. "On Fault Representativeness of Software Fault Injection". In: *IEEE Transactions on Software Engineering* 39.1 (Jan. 2013), pp. 80–96. DOI: 10.1109/TSE.2011.124.

[Nat13] Roberto Natella. *SAFE: SoftwAre Fault Emulator tool*. 2013. URL: http://wpage.unina.it/roberto.natella/tools.html.

[NB13] Ruslan Nikolaev and Godmar Back. "VirtuOS: An Operating System with Kernel Virtualization". In: *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. ACM, 2013. DOI: 10.1145/2517349.2522719.

[NC01] Wee Teck Ng and Peter M Chen. "The Design and Verification of the Rio File Cache". In: *IEEE Transactions on Computers* 50.4 (2001), pp. 322–337. DOI: 10.1109/12.919278.

[Nec+02] George C Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Conference on Compiler Construction*. 2002.

[Nie94] Jakob Nielsen. *Usability Engineering*. Academic Press Inc., 1994.

[Nov+13] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments". In: *Proc. of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. 2013, pp. 219–230.

[OU10] M. Oriol and F. Ullah. "YETI on the Cloud". In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. Apr. 2010, pp. 434–437. DOI: 10.1109/ICSTW.2010.68.

[Pad+08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. "Documenting and Automating Collateral Evolutions in Linux Device Drivers". In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys '08. Glasgow, Scotland UK: ACM, 2008, pp. 247–260. DOI: 10.1145/1352592.1352618.

[Pal+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. "Faults in Linux: Ten Years Later". In: *Proc. of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. ACM, 2011, pp. 305–318. DOI: 10.1145/1950365.1950401.

[Par+09] T. Parveen, S. Tilley, N. Daley, and P. Morales. "Towards a distributed execution framework for JUnit test cases". In: *2009 IEEE International Conference on Software Maintenance*. Sept. 2009, pp. 425–428. DOI: 10.1109/ICSM.2009.5306292.

[PFH03] Niels Provos, Markus Friedl, and Peter Honeyman. "Preventing Privilege Escalation". In: *Proc. of the 12th USENIX Security Symposium*. USENIX Security '03. USENIX Association, 2003.

[PHY11] K. Praditwong, M. Harman, and Xin Yao. "Software Module Clustering as a Multi-Objective Search Problem". In: *IEEE Transactions on Software Engineering* 37.2 (Mar. 2011), pp. 264–282. DOI: 10.1109/TSE.2010.26.

[Pip+15] Thorsten Piper, Stefan Winter, Oliver Schwahn, Suman Bidarahalli, and Neeraj Suri. "Mitigating Timing Error Propagation in Mixed-Criticality Automotive Systems". In: *Proceedings of the 2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. ISORC '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 102–109. DOI: 10.1109/ISORC.2015.13.

[Pri09] Princeton University. *The PARSEC Benchmark Suite*. 2009. URL: http://parsec.cs.princeton.edu/parsec3-doc.htm.

[QSu19a] Q-Success. *Usage of web servers for websites*. 2019. URL: https://w3techs.com/technologies/overview/web_server/all (visited on 02/11/2019).

[QSu19b] Q-Success. *Usage statistics and market share of Unix for websites*. 2019. URL: https://w3techs.com/technologies/details/os-unix/all/all (visited on 02/11/2019).

[Qua18] Qualcomm Technologies, Inc. *Qualcomm Snapdragon 845 Mobile Platform*. 2018. URL: https://www.qualcomm.com/media/documents/files/snapdragon-845-mobile-platform-product-brief.pdf (visited on 02/10/2019).

[Rod+99] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid". In: *Proc. of the Third European Dependable Computing Conference on Dependable Computing*. Ed. by Jan Hlavička, Erik Maehle, and András Pataricza. 1999, pp. 143–160.

[RS09] Matthew J Renzelmann and Michael M Swift. "Decaf: Moving Device Drivers to a Modern Language". In: *Proc. of the 2009 USENIX Annual Technical Conference*. USENIX '09. USENIX Association, 2009.

[RTw16] RTwiki Team. *Real-Time Linux Wiki*. 2016. URL: https://rt.wiki.kernel.org/index.php/Main_Page (visited on 08/01/2016).

[Rup18] Karl Rupp. *42 Years of Microprocessor Trend Data*. Feb. 15, 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data (visited on 02/12/2019).

[Rus81] J. M. Rushby. "Design and Verification of Secure Systems". In: *Proc. of the Eighth ACM Symposium on Operating Systems Principles*. SOSP '81. ACM, 1981. DOI: 10.1145/800216.806586.

[Ryd79] B.G. Ryder. "Constructing the Call Graph of a Program". In: *IEEE Transactions on Software Engineering* SE-5.3 (May 1979), pp. 216–226. DOI: 10.1109/TSE.1979.234183.

[SAM08] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. "Sufficient Mutation Operators for Measuring Test Effectiveness". In: *Proc. of the 30th international conference on Software engineering*. ICSE'08. 2008, pp. 351–360. DOI: 10.1145/1368088.1368136.

[SBK10] Daniel Skarin, Raul Barbosa, and Johan Karlsson. "Comparing and Validating Measurements of Dependability Attributes". In: *2010 European Dependable Computing Conference*. 2010, pp. 3–12. DOI: 10.1109/EDCC.2010.11.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. "Improving the Reliability of Commodity Operating Systems". In: *Proc. of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. ACM. ACM, 2003, pp. 207–222. DOI: 10.1145/945445.945466.

[SC09] Gaurav Somani and Sanjay Chaudhary. "Application performance isolation in virtualization". In: *2009 IEEE International Conference on Cloud Computing*. 2009, pp. 41–48. DOI: 10.1109/CLOUD.2009.78.

[SC13] Y. Sun and T. c. Chiueh. "SIDE: Isolated and efficient execution of unmodified device drivers". In: *Proc. of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '13. June 2013. DOI: 10.1109/DSN.2013.6575348.

[Sch+18a] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "FastFI: Accelerating Software Fault Injections". In: *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)*. PRDC'18. Taipei, Taiwan, Dec. 2018, pp. 193–202. DOI: 10.1109/PRDC.2018.00035.

[Sch+18b] Oliver Schwahn, Stefan Winter, Nicolas Coppik, and Neeraj Suri. "How to Fillet a Penguin: Runtime Data Driven Partitioning of Linux Code". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (Nov. 2018), pp. 945–958. DOI: 10.1109/TDSC.2017.2745574.

[Sch+19] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "Assessing the State and Improving the Art of Parallel Testing for C". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019. [under submission].

[Sha+03] S.C. Shaw, M. Goldstein, M. Munro, and E. Burd. "Moral Dominance Relations for Program Comprehension". In: *IEEE Transactions on Software Engineering* 29.9 (Sept. 2003), pp. 851–863. DOI: 10.1109/TSE.2003.1232289.

[Sim03] Daniel Simpson. *Windows XP Embedded with Service Pack 1 Reliability*. Jan. 2003. URL: http://msdn.microsoft.com/en-us/library/ms838661(WinEmbedded.5).aspx (visited on 08/01/2016).

[SP10] Matt Staats and Corina Păsăreanu. "Parallel Symbolic Execution for Structural Test Generation". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. 2010, pp. 183–194. DOI: 10.1145/1831708.1831732.

[Spe+06] Michael F. Spear, Tom Roeder, Orion Hodson, Galen C. Hunt, and Steven Levi. "Solving the Starting Problem: Device Drivers As Self-describing Artifacts". In: *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. ACM, 2006. DOI: 10.1145/1217935.1217941.

[SS75] J. H. Saltzer and M. D. Schroeder. "The protection of information in computer systems". In: *Proc. of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

[Sta00] E. Starkloff. "Designing a parallel, distributed test system". In: *Proc. AUTOTESTCON'00*. 2000, pp. 564–567.

[Sto+00] D.T. Stott, B. Floering, Z. Kalbarczyk, and R.K. Iyer. "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors". In: *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*. 2000, pp. 91–100. DOI: 10.1109/IPDS.2000.839467.

[Tan+07] Lin Tan, E.M. Chan, R. Farivar, N. Mallick, J.C. Carlyle, F.M. David, and R.H. Campbell. "iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support". In: *Proc. of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*. DASC '07. Sept. 2007. DOI: 10.1109/DASC.2007.16.

[Ton01] P. Tonella. "Concept Analysis for Module Restructuring". In: *IEEE Transactions on Software Engineering* 27.4 (Apr. 2001), pp. 351–363. DOI: 10.1109/32.917524.

[Tsa+99] T.K. Tsai, M.C. Hsueh, H. Zhao, Z. Kalbarczyk, and R.K. Iyer. "Stress-based and path-based fault injection". In: *IEEE Trans. on Computers* 48.11 (1999), pp. 1183–1201.

[VFI16] VFIO Maintainers. *VFIO - Virtual Function I/O*. 2016. URL: https://www.kernel.org/doc/Documentation/vfio.txt (visited on 08/01/2016).

[VM03] Marco Vieira and Henrique Madeira. "A dependability benchmark for OLTP application environments". In: *Proc. of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. 2003, pp. 742–753.

[Voa+97] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. "Predicting How Badly "Good" Software Can Behave". In: *IEEE Software* 14.4 (1997), pp. 73–83.

[Wal12] Henry M. Walker. *The Tao of Computing*. 2nd. Chapman & Hall/CRC, 2012.

[Wan17] Jiacun Wang. *Real-Time Embedded Systems*. John Wiley & Sons, Inc., 2017.

[Wat+10] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. "Capsicum: Practical Capabilities for UNIX". In: *Proc. of the 19th USENIX Security Symposium*. USENIX Security '10. USENIX Association, 2010.

[Wig97] T.A Wiggerts. "Using Clustering Algorithms in Legacy Systems Remodularization". In: *Proc. of the Fourth Working Conference on Reverse Engineering*. WCRE '97. Oct. 1997. DOI: 10.1109/WCRE.1997.624574.

[Wil+08] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. "Device Driver Safety Through a Reference Validation Mechanism". In: *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '08. USENIX Association, 2008.

[Win+13] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. "simFI: From Single to Simultaneous Software Fault Injections". In: *2013 43rd Annu. IEEE/IFIP Int. Conf. Dependable Syst. Networks*. IEEE, June 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575310.

[Win+15a] Stefan Winter, Thorsten Piper, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. "GRINDER: On Reusability of Fault Injection Tools". In: *Proceedings of the 2015 IEEE/ACM 10th International Workshop on Automation of Software Test*. AST '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 75–79. DOI: 10.1109/AST.2015.22.

[Win+15b] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. "No PAIN, No Gain?: The Utility of PArallel Fault INjections". In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 494–505. DOI: 10.1109/ICSE.2015.67.

[Yan+13] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. "A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing". In: *ACM SIGMETRICS Performance Evaluation Review* 40.4 (Apr. 2013). DOI: 10.1145/2479942.2479946.

[YC79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. 1st. Prentice-Hall, Inc., 1979.

[Yu+09] Lian Yu, Le Zhang, Huiru Xiang, Yu Su, Wei Zhao, and Jun Zhu. "A Framework of Testing as a Service". In: *2009 International Conference on Management and Service Science*. Sept. 2009, pp. 1–4. DOI: 10.1109/ICMSS.2009.5302717.

[Yu+10] Lian Yu, Wei-Tek Tsai, Xiangji Chen, Linqing Liu, Yan Zhao, Liangjie Tang, and Wei Zhao. "Testing as a Service over Cloud". In: *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*. 2010, pp. 181–188. DOI: 10.1109/SOSE.2010.36.

[Zha+14] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. "Empirically Revisiting the Test Independence Assumption". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA'14. New York, New York, USA: ACM Press, 2014, pp. 385–396. DOI: 10.1145/2610384.2610404.

[Zho+06] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. "SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques". In: *Proc. of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. USENIX Association, 2006.

[Zhu+12] Yian Zhu, Yue Li, Jingling Xue, Tian Tan, Jialong Shi, Yang Shen, and Chunyan Ma. "What Is System Hang and How to Handle It". In: *Proc. ISSRE'12*. 2012, pp. 141–150.