Aufzeichnungsbasierte Analyse von Sperren in Betriebssystemen

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Alexander Lochmann

Dortmund

2021

Tag der mündlichen Prüfung:	15.12.2021
Dekan:	Prof. Dr.-Ing. Gernot Fink
Gutachter:	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr. rer. nat. Jens Teubner

ZUSAMMENFASSUNG

Moderne Mehrkernbetriebssysteme bieten eine Vielzahl an Synchronisationsmechanismen. Sie dienen der Realisierung von feingranularem Sperren, um dem Betriebssystem wie auch den darauf laufenden Anwendungen zu erlauben, die Leistung von modernen Mehrkernprozessoren auszunutzen. Hierbei werden ganze Subsysteme, einzelne Datenstrukturen oder lediglich Teile einer Datenstruktur mit einer oder mehr Sperren abgesichert. Je vielfältiger die Mechanismen und je feingranularer das Sperren wird, desto fehleranfälliger kann ein Betriebssystem werden. Daher ist es immanent wichtig zu verstehen, wie die vorgenannten Synchronisationsmechanismen in einem Mehrkernbetriebssystem eingesetzt werden, um Synchronisationsfehler zu vermeiden.

Existierende Forschungsarbeiten in diesem Bereich befassen sich mit dem Auffinden von spezifischen Synchronisationsproblemen, wie z. B. der Detektion von Wettlaufsituationen um Speicherzugriffe. Sie detektieren Synchronisationsfehler allerdings nur im Nachhinein. Sie leiten aber keinerlei Sperren-Regeln ab, die Aussagen über das korrekte Absichern von Zugriffen machen könnten. So würden im Vorhinein Fehler vermieden.

Genau diese Lücke versucht die vorliegende Arbeit zu schließen. Daher befasst sie sich mit den Fragen, ob man a) mit der aufzeichnungsbasierten Analyse Erkenntnisse über das Synchronisationsverhalten in Mehrkehrbetriebssystemen erlangen kann, und, wie man b) mit diesen Erkenntnissen die Softwarequalität moderner Mehrkernbetriebssysteme verbessern kann.

Daraus ergeben sich folgende Forschungsbeiträge dieser Arbeit: Zunächst wird in dieser Arbeit der Entwurf des LockDoc-Ansatzes erläutert. Dieser umfasst das Aufzeichnen von Speicherzugriffen und Sperren-Operationen in einem Betriebssystemkern, während eine Arbeitslast ausgeführt wird. Daraus werden Zusammenhänge zwischen Zugriffen auf Datenstrukturen und Sperren-Operationen hergestellt. Dies lässt sich auf dreierlei Wegen nutzen: 1) Das Überprüfen der existierenden Sperren-Dokumentation, ob der Programmcode sich noch an die dokumentierten Regeln hält. 2) Das Ableiten von neuen Sperren-Regeln für verschiedene Datentypen. Aus diesen Daten lässt sich in einem weiteren Schritt eine neue Sperren-Dokumentation generieren. 3) Das Detektieren von Zugriffen, die nicht den abgeleiteten Regeln folgen. Die sogenannten Gegenbeispiele zeigen potentielle Synchronisationsfehler inkl. der Aufrufhierarchie sowie den tatsächlich gehaltenen Sperren an.

In dieser Arbeit wird der Ansatz im Rahmen von Fallstudien auf die Betriebssystemkerne von Linux und FreeBSD angewendet. Die Untersuchung erfolgt dabei nach den drei vorgenannten Zielen. Basierend auf den Untersuchung im Rahmen dieser Arbeit wurden fünf Änderungen am Linux-Kern seitens des Autors dieser Arbeit erstellt und durch die Entwicklergemeinde akzeptiert. Eine weitere Änderung wurde bereits für gut befunden, aber

noch nicht akzeptiert. Die Ergebnisse dieser Arbeit führten ebenfalls zu einer Änderung an der Sperren-Dokumentation im FreeBSD-Kern. Außerdem wurde ein Synchronisationsfehler in FreeBSD aufgedeckt.

PUBLIKATIONEN

Teile dieser Dissertation wurden zuvor in Journalen oder in Tagungsbänden von Konferenzen, Workshops oder ähnlichem veröffentlicht (in chronologischer Reihenfolge):

Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-based analysis of locking in the Linux kernel. In *Proceedings of the 14th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '19)*, New York, NY, USA, March 2019. ACM Press. doi: 10.1145/3302424.3303948. [LSBS19] (*peer-reviewed*)

Alexander Lochmann, Robin Thunig, and Horst Schirmeier. Improving Linux-kernel tests for lockdoc with feedback-driven fuzzing. In *Tagungsband des FG-BS Herbsttreffens 2020*, Bonn, 2020. Gesellschaft für Informatik e.V.z. doi: 10.18420/fgbs2020h-01. [LTS20] (*nicht peer-reviewed*)

DANKSAGUNGEN

Als Erstes möchte ich Prof. Dr. Olaf Spinczyk danken, dass er mir die Gegelegenheit gab, das Promotionsvorhaben vor einigen Jahren in seiner Arbeitsgruppe zu beginnen. Ich danke ihm auch für Anregungen und Kritik in unzähligen Gesprächen zu Papieren sowie dem Korrekturlesen dieser Arbeit. Ebenso möchte ich Prof. Dr. Jens Teubner danken. Er hat sich nicht nur freundlicherweise bereiterklärt, das Zweitgutachten anzufertigen. Vielmehr nahm er mich an seinem Lehrstuhl auf und erlaubte mir so, mein Promotionsvorhaben zu beenden.

Natürlich gilt mein Dank auch der Deutschen Forschungsgemeinschaft (DFG). Sie gab mir mit dem Sonderforschungsbereich 876 „Verfügbarkeit von Information durch Analyse unter Ressourcenbeschränkung“ (Kennzeichen SP 968/9-1) die Möglichkeit, als Mitarbeiter des Teilprojekts A1 Teile meiner Arbeit im Rahmen des Sonderforschungsbereichs durchzuführen.

Danken möchte ich sowohl den Kollegen der ehemaligen Arbeitsgruppe „Eingebettete Systemsoftware“ sowie den KollegInnen des Lehrstuhls für Datenbanken und Informationssysteme: Maximilian Berens, Christoph Borchert, Hendrik Borghorst, Markus Buschhoff, Dr. Michael Engel, Daniel Friesel, Henning Funke, Ulrich Gabor, Oliver Gasser, Andreas Grosche, Boguslaw Jablkowski, Roland Kühn, Thomas Lindemann, Matthias Meier, Jan Mühlig, Michael Müller, Dr. Horst Schirmeier, Lea Schönberger und Jochen Streicher. Danke für viele lustige und unterhaltsame Momente auf der Arbeit oder abseits der Arbeit bei dem ein oder anderen Kaltgetränk. Ich danke auch Dr. Michael Engel für unzählige Gespräche und aufmunternde Worte während meines Studiums an der TU Dortmund.

Ganz besonderer Dank gilt Dr. Horst Schirmeier. Er hat nicht nur mit seiner Arbeit an dem Projekt LockDoc zu dessen aktuellem Zustand beigetragen. Vielmehr gab er mir mit den unzähligen Diskussionen der Form „Ich schau mal eben in deinem Büro vorbei.“ wichtiges Feedback für meine Arbeit. Gerade in der Endphase meiner Dissertation hatte er immer ein aufmunterndes Wort parat.

Ohne das Feedback des Linux-Entwicklers Jan Kara wären viele Erkenntnisse und Anpassungen an den Linux-Kern nicht möglich gewesen. Jan hat stets auf E-Mails mit Fragen und Zwischenergebnissen – teils ausführlich – geantwortet. „Thx, Jan!“

Zu guter Letzt, aber nicht minder bedeutend, möchte ich meiner Familie, insbesondere meinem Vater, und meinen Freunden für ihre Unterstützung und den Rückhalt danken. Dankeschön, Maurice, für das Lektorat dieser Arbeit! Danke, Lisa und Stefan, für die Einblicke in die Welt der Mengen. Ganz besonders möchte ich meiner Partnerin, Laura, danken. Ohne ihre Unterstützung in jeglicher Form wäre diese Arbeit nicht so möglich gewesen. Danke, Laura, dass Du mich ertragen hast. Der Dank gilt aber auch meinem Sohn

Mattis. Seine gelegentlichen Besuche in meinem Arbeitszimmer während der Zeit im Home-Office sorgten immer für ein Lächeln.

INHALTSVERZEICHNIS

1 EINLEITUNG

"*Multiple cores are the new megahertz. Multicore will be the transition from brute-force performance to architectural elegance.*"

— John Williams, Advanced Micro Devices, Mai 2005

Inhalt

Moores Gesetzmäßigkeit [Moo65] in Kombination mit der von Robert H. Dennard formulierten Formel [DGR+74] zur Verkleinerung der Strukturgrößen von integrierten Schaltkreisen dienten der Industrie in den vergangenen Jahrzehnten als Vorbild, um mit jeder neuen Prozessorgeneration eine stetig steigende Leistung zu erzielen. Mit zunehmender Verkleinerung der Strukturbreiten stießen die Chip-Entwickler an physikalische Grenzen, die eine Leistungssteigerung rein mit Hilfe der Taktfrequenz pro Prozessorkern stark erschwerte [EBA+11]. Um weiter eine Leistungssteigerung zu erzielen, werden auf Kosten von einzelnen und starken Prozessorkernen eine steigende Anzahl von Prozessorkernen auf einem Chip gefertigt [Gee05, CB08, EBA+11]. Somit behielt Moores Gesetz noch lange seine Gültigkeit.

Dieser Trend hin zu *Symmetric Multithreading* führte bereits Anfang der 1990-er Jahre zu einer Anpassung der existierenden Betriebssysteme, um diese Leistung auch auszunutzen zu können [CB08]. Dabei wurde zunächst eine Sperre, die den gesamten Kern absichert, eingesetzt, um die Daten und den Zustand eines Betriebssystems gegen konkurrierende Zugriffe zu schützen [Leh01, Jon14b, CB08]. Über die Jahre wurden daraus mehrere Sperren, die Teile oder einzelne Datenstrukturen des Betriebssystemkerns absicherten [Leh01, Jon14b, Lov10, MNNW14]. Zusätzlich zu dem feingranularerem Sperren kamen mit der Zeit weitere Sperren-Typen hinzu, die einem Betriebssystementwickler zur Verfügung stehen [Lov10, MNNW14]. Somit gibt es einen umfassenden Werkzeugkasten an Sperren-Typen auf der einen Seite und viele Einsatzmöglichkeiten dieser Sperren auf der anderen Seite. In Kombination können beide Aspekte bei der Realisierung von Synchronisationsmustern im Betriebssystem zu Programmierfehlern führen. Verschiedene Studien zeigen, dass Programmierfehler in Betriebssystemen oder Anwendungssoftware, die auf fehlende oder falsche Synchronisation zurückzuführen sind, relevant sind [CYC+01, LPSZ08, AASEH17]. Welche fatalen Folgen Synchronisationsfehler im Allgemeinen haben können, lässt sich an zwei

Beispielen verdeutlichen: In den 1980er Jahren führten mehrere Synchronisationsfehler zu verschiedenen Fehlfunktionen des *Therac-25*, einem Linearbeschleuniger zur Anwendung in der Strahlentherapie [LT93]. Es konnte u.a. zu einer Wettkampfbedigung um eine Zustandsvariable zwischen dem Steuerungsprozess und dem Prozess für die Benutzereingabe kommen, während der Benutzer die Parameter eingab [LT93]. Der inkonsistente Zustand wurde ohne Überprüfung übernommen und sorgte für eine zu hohe Strahlendosis [LT93]. Aufgrund der Fehlfunktion wurden Patienten mehrfach einem Vielfachen der sonst üblichen Strahlendosis ausgesetzt – einzelne verstarben daran [LT93].

Ein weiteres Beispiel für die Auswirkungen von Synchronisationsfehlern ereignete sich im Sommer 2003: In Teilen Kanadas und den Vereinigten Staaten von Amerika kam es zu einem großflächigen, mehrstündigen Stromausfall [U.S04]. Verantwortlich dafür war auch hier ein Programmierfehler, der auf fehlerhafte Synchronisation[1] zurückzuführen ist [Nat07]. Durch die Wettlaufsituation stürzte das primäre Energieüberwachungssystem, das die Betreiber über den aktuellen Stromfluss informiert, unbemerkt ab [Nat07]. Durch die somit erschwerte Überwachung des Übertragungsnetzes kam es in Folge von weiteren Fehlern letztlich zum Stromausfall [U.S04, Nat07]. Der Ausfall der Überwachungssysteme wird als ein Bestandteil des Ausfalls genannt [U.S04, S. 51 ff.].

Wenngleich beide Beispiele Anwendungssoftware betreffen, so zeigen sie dennoch die Reichweite von Synchronisationsfehlern auf. Bei Anwendungssoftware besteht grundsätzlich die Möglichkeit, sie neu zu starten, ohne dabei das gesamte System in Mitleidenschaft zu ziehen. Werden hingegen Abläufe im Betriebssystem falsch synchronisiert, so besteht die Gefahr, dass z. B. das komplette System instabil wird oder Daten korrumpiert werden könnten. Daher ist es hierbei umso wichtiger, Fehler im Betriebssystem zu vermeiden. Die Bedeutung steigt mit der Komplexität der Betriebssysteme. Ein Blick auf Linux unterstreicht dies noch einmal: Erste Untersuchungen wurden dazu von Israeli und Feitelson durchgeführt [IF10]. Sie zeigten in ihrer Arbeit bereits, wie stark die Komplexität des Linux-Kerns von 1994 bis 2008 gestiegen ist [IF10]. In eigenen Untersuchungen konnte dieser Trend bestätigt werden: Der obere Graph in Abbildung 1.1 zeigt, dass die Codegröße[2] in den letzten 10 Jahren um 110,4 % gestiegen ist. Analog zu der Arbeit von Israeli und Feitelson wurde die Codekomplexität nach McCabe [McC76] ermittelt, welche um 85,48 % stieg. Zusätzlich ist in Abbildung 1.1 dargestellt, wie sich die Verwendung von Sperren im Linux-Kern über die Zeit entwickelten. Die Anzahl der verwendeten Sperren vom Typ *mutex* ist beispielsweise in demselben Zeitraum um 111,24 % gestiegen. Dies verdeutlicht, dass die Komplexität des Linux-Kerns bzgl. der Synchronisation zunimmt.

Eine genauere Betrachtung sowie Erklärung der Analyse findet sich in Abschnitt 3.1.

Zusätzlich zur steigenden Komplexität zeigen verschiedene Arbeiten, dass es statistisch gesehen je nach Metrik und untersuchter Software eine bestimmte Anzahl an Programmierfehlern gibt, wie z. B. die von Basili et al.,

1 https://www.theregister.co.uk/2004/04/08/blackout_bug_report/

2 Anzahl der effektiven Zeilen Code, ohne Leerzeilen und Kommentare. Ermittelt mittels *cloc* (https://github.com/AlDanial/cloc).

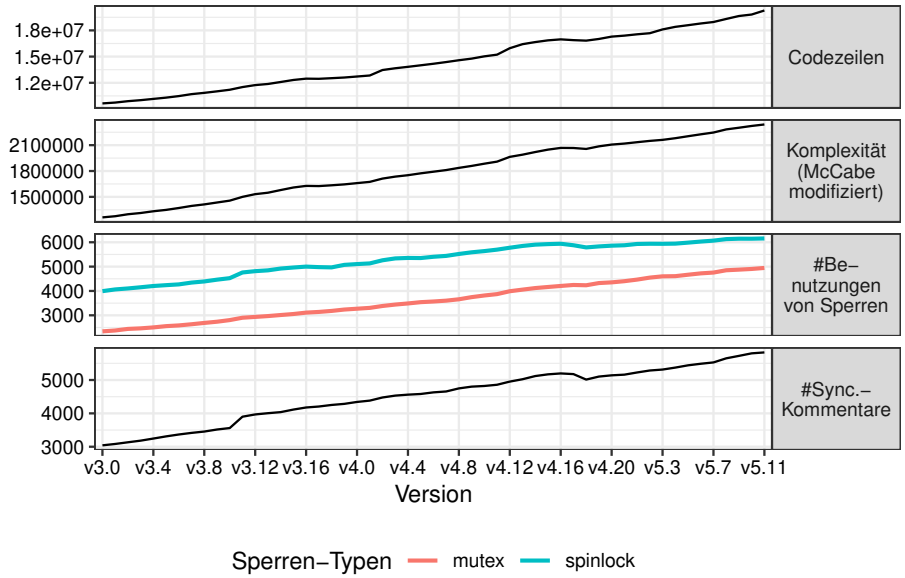


Abbildung 1.1: Die Entwicklung der Codegröße sowie der Code-Komplexität des Linux-Kernels über die letzten 10 Jahren sowie die Verwendung von Initialisierungsfunktionen von zwei verschiedenen Sperren-Typen über denselben Zeitraum. Ebenso werden die Anzahl der der synchronisations-relevanten Quellcodekommentare dargestellt. (Nach einer Vorlage von Lochmann et al. [LSBS19])

Chou et al. oder Ostrand et al. [BP84, CYC+01, OW02, OWB04]. In Summe verdeutlichen die genannten Argumente den Bedarf zur Überprüfung von Betriebssystemcode hinsichtlich einer korrekt durchgeführten Synchronisation bei Zugriffen auf gemeinsam genutzter Daten.

Die Überprüfung von Betriebssystemcode ist daher Gegenstand aktueller Forschung – wie der folgende Abschnitt noch detaillierter erläutern wird. Diese Arbeit wählt mit der aufzeichnungsbasierten Analyse eine andere Herangehensweise als bisherige Arbeiten. Statt lediglich den Programmcode zu überprüfen verfolgt diese Arbeit das übergeordnete Ziel, wie man einen generischen Ansatz zur Analyse und Verbesserung von Sperren-Dokumentation in Betriebssystemkernen entwirft. Wie genau dies umgesetzt wird, wird u.a. im Forschungsbeitrag 1 weiter ausgeführt.

1.1 FORSCHUNGSBEITRAG

Das in dieser Arbeit behandelte Themengebiet ist in der Forschung breit vertreten. Es existieren bereits verschiedenste Arbeiten, die sich mit der Analyse von Programmierfehlern im Allgemeinen und Synchronisationsfehlern im Speziellen in Betriebssystemen befassen.

An erster Stelle ist hier die formale Verifikation von Software zu nennen. Wenngleich diese Methode nicht explizit auf die Überprüfung der korrekten Synchronisation ausgerichtet ist, so kann hiermit gezeigt werden, dass eine Software korrekt funktioniert – sofern die Transformation zwischen formalem Modell und Programmcode korrekt ist [Som16, S. 299 ff., S. 356].

Somit werden Fehler von vorneherein ausgeschlossen. Im Rahmen von Betriebssystemen wurde diese Methode u.a. auf das Mikrokern-Betriebssystem *seL4*[3] erfolgreich angewendet [KEH+09]. Mikrokern-Betriebssysteme haben von Natur aus eine kleine Codebasis, welche die Verifikation erleichtert. Darüber hinaus befassen sich verschiedene Arbeiten mit der Überprüfung (von Teilen) eines Mehrzweckbetriebssystems wie z. B. Linux [Wito7, CZC+15, RST+15, dOCdO19]. Einen Gegenentwurf stellt die Betriebssystementwicklung mit der Programmiersprache *Rust* [KN19] dar: Hier ist die Sicherheit, z. B. gegenüber Wettlaufsituationen bei Speicherzugriffen, bereits Bestandteil der Sprache sowie des Übersetzers [KN19]. Somit kann zum Übersetzungszeitpunkt bereits eine Verifikation erfolgen [KN19]. Gleichzeitig bietet *Rust* den unsicheren, direkten Zugriff auf Speicher an, wie es bei der Betriebssystementwicklung erforderlich ist [KN19]. In verschiedenen Arbeiten wurde bereits die Umsetzung eines Betriebssystems in *Rust* betrachet [LAC+15, LBP19].

Ein detaillierter Überblick über existierende Arbeiten in diesem Feld finden sich in Kapitel 2.

Wie bereits erwähnt, ist das übergeordnete Ziel dieser Arbeit der Entwurf eines generischen Ansatzes zur Analyse und Verbesserung der Sperren-Dokumentation in Betriebssystemkernen. Die existierenden Arbeiten in diesem Bereich bietem dem Entwickler bereits Unterstützung: Sie finden beispielweise Wettlaufsituationen zwischen Speicherzugriffen [ECH+01, EA03, LPH+07, VAR+16, BLCH19, SBN+97, NS07, EMBO10, JYX+16, CBJ+19, Sha19], untersuchen anderweitige fehlerhafte Nutzungen von Sperren [BV04, BR02, Lino4, NS07, HCC+12, BHoo, CS12] oder detektieren Verklemmungen [EA03, BV04, NS07, AS06, JTZC08, Coro6, MNNW14, Mat11]. Dabei setzt ein Teil der Arbeiten auf statische Code-Analyse [SBN+97, ECH+01, BR02, EA03, BV04, Jono4, LPH+07, VAR+16]. Andere Arbeiten hingegen führen ihre Analyse zur Laufzeit durch [SBN+97, BHoo, AS06, Coro6, NS07, JTZC08, HCC+12, CS12, MNNW14, JYX+16, JKS+19, CBJ+19]. Besonders wichtig ist hier die Arbeit von Savage et al. [SBN+97]. Sie legen mit ihrem *LockSet*-Algorithmus einen Grundstein für spätere Arbeiten. Die letzte Gruppe erhebt zur Laufzeit nur die notwendigen Daten [LSC97, LCM+05, Mat11, ÅNK11, LCH+12, Sha19]. Die Analyse führen sie jedoch offline durch.

Meist beschränken sich die Arbeiten entweder auf eine Fehlerart, wie z. B. das Auffinden von Wettlaufsituationen, oder untersuchen nur einen Teil des jeweiligen Betriebssystems. Darüber hinaus gibt es bisher wenige Arbeiten zur Laufzeitanalyse. Dies legt die Vermutung nah, dass es wert ist, weitere Untersuchungen in dieser Richtung durchzuführen. Aus diesem Grund nähert sich diese Arbeit dem Problem mit einer aufzeichnungsbasierten Analyse. Hierbei werden die erforderlichen Daten zur Laufzeit, während das Zielsystem unter Last arbeitet, erhoben und später analysiert. Dabei widmet die Arbeit sich folgender Fragestellung:

FORSCHUNGSFRAGE 1: Kann man mittels aufzeichnungsbasierter Analyse Erkenntnisse über das Synchronisationsverhalten in Mehrkernbetriebssystemen erlangen?

3 https://sel4.systems/

FORSCHUNGSBEITRAG 1: ENTWURF UND BEWERTUNG VON LOCKDOC

Der erste und wichtigste Forschungsbeitrag bezieht sich auf den Entwurf von LockDoc. Hierbei handelt es sich um einen generischen Ansatz zur Sperren-Analyse in komplexen Softwaresystemen. Die Analyse geht dabei runter bis auf einzelne Elemente der beobachteten Datenstruktur(en). Wenngleich LockDoc auf jedes Softwaresystem anwendbar ist, lag der Fokus auf dem Einsatz in Betriebssystemen. Dabei vollzieht LockDoc folgende Schritte:

1. Während der Ausführung einer Arbeitslast werden Speicherzugriffe sowie Operationen auf Sperren innerhalb des zu untersuchenden Betriebssystems aufgezeichnet.
2. Die aufgezeichneten Daten werden in ein relationales Schema überführt und dabei mit Informationen aus dem Quelltext angereichert.

Abschließend beinhaltet dieser Forschungsbeitrag auch eine Plausibilitätsprüfung des LockDoc-Ansatzes. Hierbei wird in einem fiktivem Szenario überprüft, ob LockDoc die vorher festgelegten Sperren-Regeln korrekt ableitet sowie die eingebauten Fehler findet.

Die bisherigen Arbeiten lassen außerdem das Rückführen der gewonnen Informationen zum Entwickler vermissen: Durch das Auffinden von Fehlern helfen sie zwar, Fehler im Nachhinein zu finden und zu beheben. Allerdings wird keinerlei Unterstützung bereitgestellt, um im Vorhinein das Entstehen von Fehlern zu verhindern. Dies kann beispielsweise durch Informationen, wie z. B. „Für Datenstruktur X sind die Sperren (1) und (2) erforderlich“, erreicht werden. Genau an diesem Punkt schließt sich Forschungsfrage 2 an:

FORSCHUNGSFRAGE 2: Wie kann man die Erkenntnisse aus Forschungsbeitrag 1 nutzen, um die Softwarequalität von Betriebssystemkernen zu verbessern?

FORSCHUNGSBEITRAG 2: KONZEPTE ZUR AUSNUTZUNG DER LOCKDOC-DATEN BEI DER BETRIEBSSYSTEMENTWICKLUNG

Im Rahmen von Forschungsbeitrag 2 geht es um die Nutzung der in Forschungsbeitrag 1 gewonnenen Daten. Mit deren Auswertung lassen sich die folgenden drei Ziel erreichen:

- Das Aufstellen neuer Hypothesen für lesende wie schreibende Zugriffe auf einzelne Elemente von Datenstrukturen und somit die Generierung neuer Dokumentation für die Entwickler,
- das Überprüfen der existierenden Dokumentation und
- das Auffinden und Aufbereiten von Fehlern im Programmcode.

Auf diesem Weg werden die gewonnenen Erkenntnisse an die Entwickler zurückgeführt: Durch das Auffinden von Fehlern wird die Software un-

mittelbar verbessert. Hierbei geht es auch um eine kompakte und übersichtliche Darstellung der Fehler, so dass ein Entwickler schnell einen Überblick gewinnt.

Das Überprüfen der Dokumentation sowie das Generieren neuer Dokumentation hilft den Entwicklern, die Softwarequalität langfristig stabil zu halten. Dies erleichtert nebenbei die Einstiegshürde für Entwickler, die zum ersten Mal an einer Betriebssystemkomponente arbeiten.

Der in Forschungsbeitrag 1 vorgestellte Ansatz wurde so entworfen, dass er sich grundsätzlich auf beliebige Softwaresysteme anwenden lässt. Dennoch gilt es zu prüfen, wie gut dies in der Realität der Fall ist. Dazu zählt zum einen, wie gut die gewählten Abstraktionen die Realität abbilden, zum anderen, wie gut der Ansatz letztlich skaliert. Dies wird in der folgenden Forschungsfrage näher untersucht:

FORSCHUNGSFRAGE 3: Lässt sich der Ansatz erfolgreich auf hochkomplexe existierende Betriebssysteme anwenden?

FORSCHUNGSBEITRAG 3: BEWERTUNG VON LOCKDOC DURCH FALLSTUDIEN

Der dritte Forschungsbeitrag widmet sich der Untersuchung von verschiedenen Betriebssystemkernen mit Hilfe von LOCKDOC. Sie wurden hinsichtlich der drei Ziele, die unter Forschungsbeitrag 2 aufgelistet wurden, untersucht. Dabei wurden die Betriebssystemkerne von Linux und FreeBSD herangezogen. Für beide Betriebssysteme war es möglich, den Großteil der vorhandenen Sperren-Typen in dem Modell von LOCKDOC abzubilden. Außerdem konnten in beiden Fällen die drei Ziele verfolgt werden.

Im Fall des Betriebssystemkerns *Linux* führte die Untersuchungen im Rahmen dieser Arbeit bereits zu fünf Änderungen am Programmcode, die von dem Autor dieser Arbeit eingereicht und von der Entwicklergemeinschaft akzeptiert wurden. Darunter befindet sich ein Programmierfehler. Außerdem wurden in einer Änderung die Sperren-Dokumentation von zwei Elementen aufgrund der Ergebnisse von LOCKDOC angepasst. Für FreeBSD führten die Ergebnisse dieser Arbeit zu zwei Änderungen. Eine Änderung betrifft die Sperren-Dokumentation. Die andere Änderung sichert einen unsynchronisierten Zugriff ab.

1.2 GLIEDERUNG

Im Folgenden wird ein Überblick über die Struktur dieser Arbeit gegeben sowie die wichtigsten inhaltlichen Punkte der jeweiligen Kapitel dargestellt:

KAPITEL 2: Hier werden die Grundlagen der für diese Arbeit relevanten Themen vorgestellt. Dazu wird zunächst in Abschnitt 2.1 eine Zusammenfassung über die zwei für diesen Kontext relevanten Betriebssystemkernmodelle gegeben. Anschließend folgt in Unterabschnitt 2.2.1

ein Überblick über die verschiedenen Arten von Sperren in der Literatur. Ferner werden die Auswirkungen von falscher Synchronisation behandelt.

KAPITEL 3: Anhand der beiden monolitischen Betriebssystemkerne von Linux und FreeBSD wird der aktuelle Zustand der Sperren-Dokumentation dargestellt. Dies beinhaltet eine Darstellung der Komplexität der Betriebssystemkerne wie auch die Vielfalt der vorhandenen Synchronisationsmechanismen.

KAPITEL 4: Es werden die existierenden Arbeiten aus diesem und angrenzenden Forschungsbereichen beleuchtet. Dabei wird nach drei Kategorien unterschieden: In Abschnitt 4.1 werden Arbeiten zur statischen Code-Analyse betrachtet, die sich mit dem Auffinden von Programmierfehlern befassen. Abschnitt 4.2 widmet sich Verfahren zum Auffinden von Synchronisationsfehlern zur Laufzeit. Zuletzt werden Arbeiten zur nachträglichen Analyse von Softwaresystemen vorgestellt. Den Abschluss bildet eine Analyse des aktuellen Stand der Kunst, die der Einordnung dieser Arbeit in den wissenschaftlichen Kontext dient.

KAPITEL 5: Hier wird der LockDoc-Ansatz sowie die zugrundeliegenden Annahmen vorgestellt. Hierbei wird dargelegt, warum ein aufzeichnungsbasierter Ansatz gewählt wurde und welche Rolle dabei die gewählte Arbeitslast spielt. Zusätzlich wird ein Ansatz zur Verbesserung der Arbeitslast aufgezeigt. Im weiteren Teil des Kapitels wird die Analyse der aufgezeichneten Daten erläutert. Dies umfasst insbesondere das Aufstellen von Sperren-Regeln sowie das Auswählen einer Sperren-Regel als Gewinner, um ein bestimmtes Datum abzusichern. Hierzu werden verschiedene Auswahlstrategien vorgestellt.

KAPITEL 6: Dieses Kapitel beleuchtet die technischen Aspekte des LockDoc-Ansatzes. Hierzu zählt die Realisierung der Aufzeichnung und das Bestimmen der Sperren-Regeln. Des weiteren werden die drei Ergebnisse des LockDoc-Ansatzes vorgestellt: 1) Das Überprüfen von existierender Sperren-Dokumentation, 2) das Generieren von neuer Sperren-Dokumentation und 3) das Auffinden von falsch-synchronisierten Zugriffen, den sog. Gegenbeispielen.

KAPITEL 7: Zunächst befasst sich Kapitel 7 mit der Plausibilitätsprüfung des Ansatzes selbst, wie es in Forschungsbeitrag 1 dargelegt wurde. Im weiteren Verlauf von Kapitel 7 werden zwei Fallstudien vorgestellt. Hierbei handelt es sich um die Anwendung von LockDoc auf die Betriebssystemkerne von Linux und von FreeBSD. In beiden Fällen wird betrachtet, wie sich die verschiedenen Sperren der jeweiligen Betriebssysteme in LockDoc abbilden lassen. Zusätzlich werden die vorgestellten Auswahlstrategien evaluiert. Für Linux wird darüber hinaus der Ansatz zur Verbesserung der Arbeitslast untersucht. Zum Abschluss werden die eigentlichen Ergebnisse von LockDoc für die beiden Betriebssysteme, wie sie in Kapitel 6 erläutert wurden, dargelegt.

KAPITEL 8: Abschließend werden die wichtigsten Ergebnisse sowie die Einschränkungen dieser Arbeit nochmal zusammengefasst und ein Ausblick auf zukünftige Forschungsarbeiten gegeben.

1.3 BEITRAG DES AUTORS ZU DIESER DISSERTATION

Gemäß §10 Absatz 2 der „Promotionsordnung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011“ müssen in einer Dissertation, soweit der Autor wissenschaftliche Ergebnisse verwendet, die in Kooperation mit anderen Wissenschaftlern entstanden sind, die Eigenanteile an diesen Ergebnissen aufgelistet werden. In den folgenden Abschnitten werden diese Anteile kapitelweise beschrieben.

KAPITEL 1: Die Analyse der Sperren-Benutzungen für Linux basiert auf einer ähnlichen Analyse von Hendrik Borghorst aus einer Veröffentlichung [LSBS19]. Die initiale Idee zu der Analyse sowie die in dieser Arbeit vorgestellten, erweiterten Form stammen beide von mir – sowohl für Linux als auch für FreeBSD.

KAPITEL 4: Teile der Analyse der verwandten Arbeiten wurden bereits in einem Papier veröffentlicht [LSBS19].

KAPITEL 5: Den Anstoß zu diesem Projekt gab Olaf Spinczyk. Die Ausarbeitung des konkreten Ansatzes ist aus Diskussionen zwischen Olaf Spinczyk, Horst Schirmeier und mir hervorgegangen und mündeten in einem Papier [LSBS19]. Die Darstellung über Mengen und das Modell der abbildbaren Sperren stammen von mir. Das Konzept der Transaktionen stammt von Horst Schirmeier [LSBS19]. Die Idee zur Betrachtung der Beziehung zwischen den Hypothesen als Graph wurde gleichermaßen von Olaf Spinczyk und Horst Schirmeier vorgeschlagen. Die Idee zu der Strategie *Sharpen* stammt von Horst Schirmeier. Die Ausarbeitung der Ideen sowie die Implementierung wurden von mir durchgeführt – ebenso deren Evaluation. Die Evaluation der verschiedenen Arbeitslasten wurde von mir konzipiert und umgesetzt – ebenso wie die Anpassung der Tests aus dem „*Linux Test Project*“[4]. Die Modifikationen an dem Werkzeug *syzkaller*[5] wurden von der studentischen Hilfskraft Robin Thunig unter meiner Anleitung durchgeführt. Die weitergehende Analyse der Code-Abdeckung wurde von Horst Schirmeier und mir durchgeführt und mündete in einem Papier [LTS20]. Ich habe dabei die Erhebung der Daten und Horst Schirmeier die Werkzeuge zur Analyse und Visualisierung implementiert. Die zu dem Thema vorgestellten verwandten Arbeiten stammen aus dem vorgenannten Papier und wurden von Horst Schirmeier zusammengetragen [LTS20].

4 https://github.com/linux-test-project/ltp
5 *https://github.com/google/syzkaller*

KAPITEL 6: Die Komposition der Arbeitslast innerhalb der virtuellen Maschine habe ich erarbeitet. Das Konzept der Datenerhebung im Gastbetriebssystem wurde von mir ebenfalls erdacht und implementiert. Ferner habe ich das Aufzeichnen der Daten mittels des Fehlerinjektionswerkzeugs FAIL* [SHD+15] umgesetzt. Dabei unterstütze Horst Schirmeier, als der Hauptentwickler ebenjenes Werkzeugs, bei der Implementierung mit Tipps und Performanzoptimierungen. Das Werkzeug *locking-rule derivator* zur Bestimmung der Hypothesen wurde von Horst Schirmeier erarbeitet und implementiert. Ich habe es um die zusätzlichen Auswahlstrategien und die Darstellung als Graphen erweitert. Die Auswertung der Daten (Unterabschnitt 6.4.1, Unterabschnitt 6.4.2 und Unterabschnitt 6.4.3) wurde von mir umgesetzt. Außerdem habe ich die Darstellung der Berichte über mögliche Programmfehler erarbeitet und umgesetzt. Dazu wurden sie mit Hilfe von Rückmeldungen durch den Linux-Entwickler Jan Kara entsprechend von mir verfeinert. Weite Teile dieses Kapitels wurden bereits in einem Papier veröffentlicht [LSBS19].

KAPITEL 7: Die Portierung von LOCKDOC auf die Betriebssystemkerne von Linux und FreeBSD wurden von mir vorgenommen, ebenso wie die Durchführung der Fallstudien. Die wichtigsten Ergebnisse zu Linux wurden bereits veröffentlicht [LSBS19], wobei Horst Schirmeier bei der Datenauswertung beteiligt war. Die Ergebnisse zur Verbesserung der Arbeitslast (Unterabschnitt 7.2.1) stammen von Horst Schirmeier und mir – siehe vorherigen Absatz zu Kapitel 5 [LTS20].

2 GRUNDLAGEN

„It was a great aid then; now it is a scalability burden."

— Robert Love über das *Big Kernel Lock*

Inhalt

Für die vorliegende Arbeit ist das Verständnis von zwei grundlegenden Themen von Bedeutung: Betriebssystemarchitekturen sowie Synchronisationsmechanismen mit dem Fokus auf Betriebssysteme. Abschnitt 2.1 stellt dazu zwei Betriebssystemarchitekturen vor. Abschnitt 2.2 widmet sich hingegen dem zweiten Thema, den Synchronisationsmechanismen.

2.1 BETRIEBSSYSTEMKERNE

Über die vergangenen Jahrzehnte entstanden verschiedene Betriebssystemarchitekturen [TB14a]. Dazu zählen z. B. Exokernel [EKO95], Mikrokernel [Lie95, Lie96], Multikernel [BBD+09] sowie monolithische Betriebssystemkerne [TB14a]. In den folgenden Abschnitten sollen die zwei am häufigsten anzutreffenden Betriebssystemarchitekturen, die für diese Arbeit relevant sind, in ihren wichtigen Grundzügen skizziert werden. Dies dient der besseren Einordnung der Problembeschreibung respektive Entwurfsentscheidungen im Rahmen dieser Arbeit, die in Kapitel 3 und Kapitel 6 behandelt werden.

2.1.1 *Monolithische Betriebssystemkerne*

Monolithische Betriebssysteme sind heute weit verbreitet auf Allzweckcomputern und Servern [TB14a]. Sie zeichnen sich dadurch aus, dass viele Betriebssystemdienste im Kern laufen [TB14a]. Unter einem Betriebssystemkern ist das Stück Programmcode zu verstehen, das die essentiellen Funktionen bereitstellt und während der gesamten Laufzeit im Arbeitsspeicher liegt [TB14a]. Wie in Abbildung 2.1 auf der linken Seite der Abbildung zu sehen ist, zählen zu den essentiellen Komponenten die Prozessverwaltung, Interprozesskommunikation, Adressraumverwaltung genauso wie Speicherverwaltung, Gerätetreiber oder Dateisystemtreiber [TB14a]. Alle Komponenten

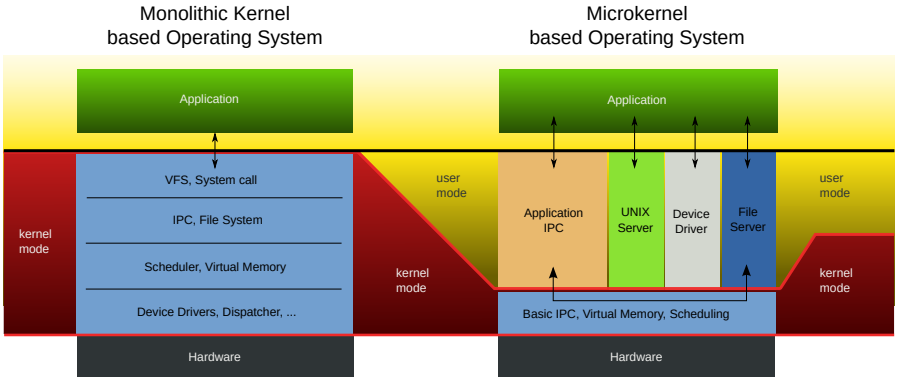


Abbildung 2.1: Vergleich zwischen der Architektur von Mikrokern- und monolithischen Betriebssystemen. (Basiert auf einer Grafik von Wikipedia-Nutzer *Golftheman* – Lizensiert nach *Public Domain)*

teilen sich hierbei einen Adressraum [Sta11]. Sie interagieren untereinander via einfachen Funktionsaufrufen, die dementsprechend auch die Schnittstellen untereinander darstellen [TB14a]. Dies erlaubt eine schnelle Abarbeitung von Systemaufrufen durch die Anwendung, bei denen mehrere Komponenten beteiligt sind. Jede Komponente kann die Schnittstelle einer jeder anderen Komponente direkt aufrufen, auf beliebige Hardware und beliebigen Speicher zugreifen [TB14a]. Hier erfolgt keinerlei Autorisierung des Aufrufers [TB14a]. Diese erfolgt lediglich beim Übertritt von der Anwendung in den Kern [TB14a]. Da der übersetzte Quellcode zu einem Programm gebunden wird, muss bei jeder Änderung der komplette Kern erneut übersetzt und gebunden werden [Sta11]. Dies ist z. B. beim Hinzufügen eines weiteren Gerätetreibers der Fall, was die Flexibilität einschränkt [Sta11]. Ferner erhöht es die Größe des resultierenden Programms, da alle nötigen Komponenten enthalten sein müssen.

Wenngleich Linux als ein Vertreter von monolithischen Betriebssystemen gilt, so liegt nicht immer der vollständige Programmcode von allen Funktionalitäten im Arbeitsspeicher [Sta11]. Vielmehr können einzelne Komponenten, wie z. B. ein Dateisystemtreiber, in von Form von Modulen zur Laufzeit nachgeladen werden [Sta11]. Außerdem können Module nachträglich übersetzt werden, ohne dass der bereits laufende Kern neugestartet werden muss [Lov10]. Dies gibt dem Linux-Kern eine gewisse Flexibilität und erlaubt es außerdem, die Größe unter Kontrolle zu behalten [Lov10].

Aus den o.g. Eigenschaften ergeben sich jedoch einige Herausforderungen: Eine fehlerhafte Komponente kann sich auf den gesamten Betriebssystemkern auswirken [TB14a]. Im schlimmsten Fall führt beispielsweise ein fehlerhafter Gerätetreiber zum Absturz des gesamten Kerns [TB14a]. Oder ein Gerätetreiber greift fälschlicherweise auf das falsche Gerät zu [TB14a].

Eine weitere Herausforderung bei monolithischen Betriebssystemen auf modernen Multiprozessorsystemen ist der durch die Anwendungen erreichbare Grad der Parallelität [Lov10]. Damit jeder Prozessorkern möglichst unabhängig von den anderen Prozessorkernen ein Programm ausführen kann, muss es mehreren Kontrollflüssen gestattet sein, gleichzeitig den Betriebs-

systemkern zu betreten [Lov10]. Hierzu muss im Kern ein feingranulares Sperren implementiert werden [Lov10]. Damit der Aufwand des Sperrens durch den Einsatz zu vieler Sperren nicht Überhand nimmt, gilt es hier, einen sinnvollen Weg zu finden [Lov10]. Die Implikationen resultierend aus dem Grad des Sperrens skizziert Unterabschnitt 2.2.2.2 detaillierter.

2.1.2 *Mikrokerne*

Bei einem Mikrokernbetriebssystem wird im Gegensatz zu einem monolithischen Betriebssystemkern der Kern auf die minimal nötigen Komponenten beschränkt [Lie95]. Dies umfasst lediglich – wie in Abbildung 2.1 verdeutlicht – eine Kontrollfluss- und Adressraumverwaltung sowie die Interprozesskommunikation [Lie95]. Alle weiteren Komponenten werden als separate Prozesse – auch Server genannt – auf die Anwendungsebene (vgl. Abbildung 2.1) ausgelagert [Lie95]. So werden Geräte- und Dateisystemtreiber, aber auch die Speicherverwaltung durch eigene Anwendungskontrollflüsse realisiert [Lie95]. Selbst die Unterbrechungsbehandlung findet in den entsprechenden Anwendungen statt. Lediglich eine minimale Behandlung erfolgt durch den Kern [Lie95]. Der Informationsaustausch zwischen den Komponenten erfolgt nicht über Funktionsaufrufe, sondern über das Versenden und Empfangen von Nachrichten mittels Interprozesskommunikation [Lie95]. So sendet eine Benutzeranwendung in *MINIX3*[1] [TW05] beispielsweise eine Nachricht an einen Server, um einen bestimmten POSIX-Systemaufruf durchzuführen [TB14a]. Auf demselben Weg erhalten Anwendungen Zugriff auf bestimmte Geräte oder Speicherbereiche [TB14a]. Der Betriebssystemkern kann so ebenfalls gewährleisten, dass auch nur derjenige Treiber auf ein Gerät respektive Speicherbereich zugreift, der auch über die nötigen Rechte verfügt [TB14a]. Als Beispiel für eine Komponente, die eine Betriebssystemfunktion implementiert, sei der externe Pager des *MACH*-Mikrokerns genannt [Lie96]. Tritt in irgendeiner Anwendung ein Speicherzugriffsfehler auf, so wird aus dieser Anwendung heraus eine Nachricht an den Pager geschickt, der den Seitenfehler behandelt [Lie96].

Da die Interprozesskommunikation ein essentieller Bestandteil von Mikrokernen ist, ist die Geschwindigkeit des Nachrichtenaustauschs entscheidend für die Performanz des gesamten Systems [Lie93, Lie96, HHL+97]. Daher liegt darauf ein Augenmerk bei der Erforschung von Mikrokernen [Lie93, Lie96, HHL+97].

Aufgrund der beschriebenen Systemarchitektur ist ein Mikrokern weniger anfällig für fehlerhafte Software. Da die meiste Funktionalität in Anwendungen ausgelagert ist, führt ein Fehlverhalten maximal zu einem Absturz der betroffenen Anwendung [TB14a]. Diese kann vom System neu gestartet werden, ohne dass der Rest des Systems oder gar der Kern in Mitleidenschaft gezogen wird [TB14a].

1 http://www.minix3.org

2.2 SYNCHRONISATIONSMECHANISMEN

In einem Uniprozessorsystem können verschiedene Kontrollflüsse, Prozesse oder Threads, nebenläufig ausgeführt werden [Sta11]. Zusätzlich können asynchrone Ereignisse wie z. B. Unterbrechungen auftreten. Deren Abarbeitung kann schlicht als weiterer Kontrollfluss betrachtet werden [Sta11].

In Multiprozessorsystemen hingegen können mehrere Kontrollflüsse sowohl verschachtelt auf einem Prozessorkern als auch überlappend auf mehreren Prozessorkernen parallel ausgeführt werden [Sta11]. Natürlich können auch hier asynchrone Ereignisse auftreten, die ebenfalls nebenläufig abgearbeitet werden.

Teilen sich nun zwei oder mehr Kontrollflüsse Daten z. B. in Form einer Liste, so lässt sich der Programmcode eines jeden Kontrollflusses in zwei Arten von Abschnitte unterteilen: Die sog. nicht-kritischen und kritischen Abschnitte[2] [Dij65, Lam86a, Sta11]. Letzterer bezeichnet den Teil des Programmcodes, der auf den geteilten Daten operiert [Dij65, Lam86a, Sta11]. Es ist dabei unerheblich, ob unter dem Begriff „Programmcode“ Anweisungen einer Hochsprache oder Maschineninstruktionen zu verstehen sind. Entscheidend ist nur, dass die gewünschte Operation auf den geteilten Daten von dem Prozessor nicht atomar ausführbar ist.

Aufgrund der o. g. möglichen Ausführungen der beschriebenen Kontrollflüsse ist nicht vorhersagbar, in welcher Reihenfolge sie ausgeführt werden. Noch viel wichtiger ist, dass ebenfalls nicht absehbar ist, zu welchen Zeitpunkten ein Kontrollfluss unterbrochen wird und ein anderer Kontrollfluss weiter arbeitet [Sta11]. Beides hängt u.a. von dem Zeitverhalten der Unterbrechungen, der Prozessablaufstrategie und dem Prozessortakt ab [Sta11]. Daher ist ebenso wenig absehbar, ob nicht mehrere Kontrollflüsse gleichzeitig ihren kritischen Abschnitt betreten. Das Ergebnis des daraus folgenden unkoordinierten Zugriffs auf die geteilten Daten hängt vom Zeitverhalten des Betriebssystems ab. Dies kann zu einer Korruption der Daten führen. Diese Situation ist unter dem Begriff *Race Condition*[3] oder *Data Race* bekannt [NM92, Sta11]. Adve et. al formalisierten die vorgenannten Abläufe in ihrer Arbeit mit der sog. *happens-before*-Relation für Speicherzugriffe durch mehrere Kontrollflüsse [AHMN91]. Mit ihrer Hilfe wird eine partielle Ordnung auf einer Menge von Speicheroperationen hergestellt und entschieden, ob eine Wettlaufsituation vorliegt [AHMN91].

Damit die geteilten Daten nach der Ausführung eines jeden kritischen Abschnitts stets in einem konsistenten Zustand sind, müssen die einzelnen kritischen Abschnitte der Kontrollflüsse sequentiell abgearbeitet werden [Lam86b, Sta11]. Die Reihenfolge, in der die kritischen Abschnitte ausgeführt werden, spielt hier keine Rolle [Lam86b, Sta11]. Auf diese Weise ist gegenseitiger Ausschluss[4] gewährleistet, eine notwendige Bedingung, damit ein Kontrollfluss nicht mit einem anderen interferiert [Dij65, Lam86b, Sta11].

2 Deutsch für *Critical Section*
3 Engl. für Wettlaufsituation
4 Deutsch für *mutual exclusion*

Der Unterabschnitt 2.2.1 gibt einen Überblick über die verschiedenen Synchronisationsprimitiven aus der Literatur. Anschließend zeigt der Unterabschnitt 2.2.2 auf, welche Herausforderungen sich bei deren Verwendung ergeben.

2.2.1 *Synchronisationsprimitiven*

In der Literatur werden verschiedene Mechanismen zur Synchronisation von nebenläufigen Kontrollflüssen vorgestellt [SGG10, Sta11, TB14b]. Darüber hinaus gibt es in der Literatur unterschiedliche Anwendungsszenarien für ebendiese Mechanismen [Ree04, SGG10, Sta11, TB14b, Dow16], wie z. B. die speisenden Philosophen [Sta11] oder das Leser-Schreiber-Problem [Sta11]. In diesem Abschnitt soll der Fokus auf die Synchronisationsprimitiven zur Realisierung von gegenseitigem Ausschluss innerhalb eines Betriebssystems gelegt werden. Dabei beschränkt sich die Betrachtung der einzelnen Mechanismen auf die konzeptionelle Sicht, die auch ein Betriebssystemprogrammierer einnehmen würde. Die Implementierung der jeweiligen Primitiven spielt hierbei keine Rolle, sofern sie nicht essentieller Bestandteil des Mechanismus ist, vgl. z. B. Spinlock. Implizite Synchronisationsmechanismen, wie z. B. Speicherbarrieren [Lov10], sind nicht Bestandteil dieses Abschnitts, da sie für die Inhalte dieser Arbeit nicht relevant sind.

2.2.1.1 *Semaphor*

Ein Semaphor besteht aus einer nicht-negativen ganzen Zahl, auf der zwei atomare Operationen definiert sind: Die Operation *v()* inkrementiert einen Semaphor um 1. Die Operation *p()* dekrementiert die Zahl um 1, sofern sie größer 0 ist [Dij68b]. Ist dies nicht der Fall, muss der aufrufende Kontrollfluss mit dem Abschluss der Operation warten bis ein anderer Kontrollfluss den Semaphor erhöht hat [Dij68b]. Bei mehreren wartenden Kontrollflüssen lässt Dijkstra zunächst offen, welcher Kontrollfluss nach einer *v()*-Operation seine Operation abschließen kann [Dij68b]. Solch ein Semaphor wird auch allgemeiner Semaphor[5] genannt [Dij68b]. An dieser Stelle abstrahiert Dijkstra von der konkreten Implementierung [Dij68b].

In heutigen Betriebssystemen werden die beiden Operationen *p()* und v() wie folgt realisiert: Die Operation *v()* erhöht den Semaphor um 1, sofern kein Kontrollfluss wartet [TB14a]. Anderenfalls wird einer der Kontrollflüsse ausgewählt und fortgesetzt [TB14a]. Der Semaphor bleibt hierbei bei 0 [TB14a]. Die Operation *p()* dekrementiert den Semaphor, sofern dieser größer 0 ist [TB14a]. Ansonsten wird der aufrufende Kontrollfluss blockiert.

In der Literatur gibt es verschiedenste Anwendungsbeispiele für sog. zählende Semaphore [Ree04, SGG10, Sta11, TB14b, Dow16]. Ein populäres Beispiel ist das Erzeuger-Verbraucher-Problem [Dij68b]. Dabei wird der Zugriff auf einen N-elementigen Puffer zwischen Erzeuger und Verbraucher derartig synchronisiert, so dass weder aus einem leeren Puffer gelesen noch in

5 Deutsch für *General Semaphore* [Dij68b]

```
1  Semaphor belegt = 0, frei = N;
2
3  void erzeuger() {
4      while (1) {
5          // Erzeuge neues Element
6          p(frei);
7          // Füge neues Element in Puffer ein
8          v(belegt);
9      }
10 }
11
12 void verbraucher() {
13     while (1) {
14         p(belegt);
15         // Entferne ein Element aus Puffer
16         v(frei);
17         // Konsumiere Element
18     }
19 }
20
21 int main() {
22     // Erzeuge Puffer der Groesse N
23     // Erzeuge 2 Kontrollfluesse mit erzeuger()/verbraucher()
24 }
```

Listing 2.1: Exemplarische Synchronisation des Erzeuger-Verbraucher-Problems mit einem N-elementigen Puffer und Semaphoren [Dij68b, Sta11]. Es ist sichergestellt, dass weder aus einem leeren Puffer gelesen noch in einen vollen Puffer geschrieben wird [Dij68b, Sta11]. (In Anlehnung an ein Beispiel aus „Operating Systems Internals and Design Principles“[Sta11, S. 225])

einen vollen Puffer geschrieben wird [Dij68b, Sta11]. Der Zugriff auf den Puffer selbst soll als atomar angenommen werden. Wie der Listing 2.1 zu entnehmen ist, werden zwei Kontrollflüsse erzeugt, die periodisch ein Element erzeugen und in den Puffer ablegen sowie ein Element aus dem Puffer entnehmen und konsumieren. Die Semaphore *frei* und *belegt* repräsentieren die Anzahl an belegten bzw. freien Elementen in dem Puffer – vgl. Zeile 1. Durch das gekreuzte Belegen bzw. Freigeben des Semaphors kann der Erzeuger nun kontinuierlich neue Elemente erzeugen und in dem Puffer ablegen, bis dieser voll ist und der Erzeuger in Zeile 6 blockiert. Andersherum kann der Verbraucher gemäß des aktuellen Werts des *belegt* Semaphors solange Elemente entnehmen, bis der Puffer leer ist und er nun selbst blockiert – vgl. 14 in Listing 2.1.

Zur Realisierung des gegenseitigen Ausschlusses wird als Spezialisierung des allgemeinen Semaphors der sog. binäre Semaphor benötigt [Dij68b]. Ein Beispiel für eine Verwendung eines binären Semaphors liefert Listing 2.2. Hier führen nun N konkurrierende Kontrollflüsse den in der Funktion *arbeit()* in Listing 2.2 dargestellten Programmcode aus, so kann nur ein Kontrollfluss in den kritischen Abschnitt gelangen. Da die Operationen auf einem Semaphor nur atomar ausgeführt werden können, gelingt es nur genau

```
1  Semaphor mutex = 1;
2
3  void arbeit() {
4      while (1) {
5          p(mutex);
6          // Kritischer Abschnitt
7          v(mutex);
8      }
9  }
10
11 int main() {
12     // Erzeuge N Kontrollfluesse mit arbeit()
13 }
```

Listing 2.2: Beispiel zur Absicherung eines kritischen Abschnitts mit einem binären Semaphor. Der Wertebereich der Semaphore liegt zwischen 0 und 1. Es kann nur ein Kontrollfluss den kritischen Abschnitt betreten [Dij68a]. (In Anlehnung an ein Beispiel aus „Operating Systems Internals and Design Principles“[Sta11, S. 218])

einem Kontrollfluss, sie erfolgreich auszuführen. Alle anderen $N - 1$ Kontrollflüsse blockieren in Zeile 5.

Ein weiteres populäres Anwendungsbeispiel ist das Leser-Schreiber-Problem: Dabei wird zwischen lesenden und schreibenden Kontrollflüssen unterschieden [Sta11]. Nicht selten gibt es hierbei viele Leser und wenige Schreiber, die auf die Daten zugreifen wollen [Sta11]. Würden alle Kontrollflüsse bei der Synchronisation gleich behandelt, so käme es zu einem Andrang bei den Lesern, da diese nur sequentiell Zugriff erhielten [Sta11]. Um dieses Problem zu vermeiden, wird auch bei der Synchronisation zwischen Lesern und Schreibern differenziert [Sta11]. Für eine funktionierende Synchronisation stellt Wiliam Stallings die drei folgenden Bedingungen auf: 1) Eine beliebige Anzahl von Lesern darf gleichzeitig auf den geteilten Daten operieren [Sta11]. 2) Zu jedem Zeitpunkt darf lediglich ein Schreiber Zugriff auf die Daten erhalten [Sta11]. 3) Wenn ein Schreiber auf den Daten operiert, darf kein Leser auf die Daten zugreifen [Sta11]. Dieses Problem lässt sich beispielsweise mit Semaphoren lösen [Sta11]. Moderne Betriebssysteme bieten allerdings Synchronisationsmechanismen an, die genau auf dieses Szenario abzielen [Lov10]. Die sogenannten Leser-Schreiber-Sperren sind in zwei Teile unterteilt: die Leser- und die Schreiber-Sperren [Lov10]. Die Leser-Sperre erlaubt es beliebig vielen Kontrollflüssen, den kritischen Abschnitt zu betreten und die Daten zu lesen, solange es keinen Schreiber gibt [Lov10]. Belegt ein schreibender Kontrollfluss die Schreiber-Sperre, so müssen alle anderen Leser warten [Lov10]. Sowohl Linux als auch FreeBSD stellen verschiedene Typen von Leser-Schreiber-Sperren bereit, wie z. B. `rwlock_t` in Linux [Lov10] oder aber die Sperre `sx` in FreeBSD.

2.2.1.2 *Mutex*

Der Mutex ist aus konzeptioneller Sicht ein binärer Semaphor, wie er bereits in Unterabschnitt 2.2.1.1 eingeführt wurde [Dij68b, Sta11]. Anstatt *p()* und *v()* heißen die Operationen *lock()* und *unlock()* [Sta11]. Die Semantik ist jedoch die Gleiche [Sta11]. Die Operation *lock()* ist ebenfalls blockierend. Darüber hinaus hat ein Mutex noch einen Besitzer, von dem er wieder freigegeben werden muss [Sta11]. Das heißt, dass ein Kontrollfluss, der ein Mutex angefordert hat, diesen auch wieder freigeben muss [Sta11].

2.2.1.3 *Spinlock*

Ein Spinlock ist aus konzeptioneller Sicht ebenfalls ein binärer Semaphor [Sta11]. Die verfügbaren Operationen heißen hier ebenfalls *lock()* und *unlock()*. Falls ein Spinlock bereits belegt ist, wird anstatt zu blockieren aktiv gewartet [Sta11]. Hierbei wird fortlaufen überprüft, ob die Sperre verfügbar, und gleichzeitig versucht, sie zu belegen [Sta11]. Daraus ergibt sich auch der Name dieses Typs, der sich von dem englischen Wort *spinning*[6] ableitet.

2.2.1.4 *Monitor*

Die bisher vorgestellten Synchronisationsprimitiven erfordern eine explizite Synchronisation: Der Programmierer muss selbst an den geeigneten Stellen die jeweiligen Aufrufe platzieren. Er muss ebenfalls sicherstellen, dass die Anzahl der Aufrufe von *p()*/*lock()* bzw. von *v()*/*unlock()* gleich sind. Um diesen fehleranfälligen Prozess [Sta11] zu umgehen, bieten Monitore ein in die Programmiersprache integriertes Konzept zur Umsetzung von gegenseitigem Ausschluss an. Eine Programmiersprache, die dieses Konzept z. B. anbietet, ist JAVA [7]. Das erstmals von C. A. R. Hoare vorgestellte Konzept soll im Folgenden erläutert werden. Ergänzungen, wie sie z. B. Lampson und Redell [LR80] entwickelt haben, gehen über den Fokus dieser Arbeit hinaus und werden nicht betrachtet.

Ein Monitor als Bestandteil einer Programmiersprache erlaubt einem Programmierer, die betreffenden Daten und die auf ihnen arbeitenden Funktionen in einem Modul zu kapseln [Hoa74, Sta11]. Dabei sorgt der Monitor implizit für einen synchronisierten Zugriff und stellt so den gegenseitigen Ausschluss sicher [Hoa74, Sta11]: Ein Kontrollfluss betritt den Monitor von außen implizit durch den Aufruf einer der im Monitor abgelegten Funktionen [Hoa74, Sta11]. Alle anderen Kontrollflüsse, die ebenfalls Funktionen aus diesem Modul aufrufen wollen, blockieren solange, bis der jeweils aktive Kontrollfluss seine Arbeit beendet hat [Hoa74, Sta11]. Muss der aktuelle Kontrollfluss jedoch auf das Eintreten einer Bedingung warten, wie z.B. dass ein Puffer nicht mehr leer ist, die nur durch die Ausführung einer anderen Funktion eintreten kann, so blockiert er mittels der Operation *wait()* auf einer sogenannten Bedingungsvariablen[8] [Hoa74, Sta11]. Dadurch wird

6 Engl. für „sich drehen“

7 https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

8 Deutsch für *condition variable*.

```
1  Mailbox box;
2
3  void arbeit() {
4      Message msg;
5      while (1) {
6          receive(box, msg);
7          // Kritischer Abschnitt
8          send(box, msg);
9          // Restliche Arbeit
10     }
11 }
12
13 void main() {
14     erzeuge_mailbox(box);
15     send(box, NULL);
16     // Erzeuge N Kontrollflüsse mit arbeit()
17 }
```

Listing 2.3: Absicherung eines kritischen Abschnitts mittels Message Passing (Adaptiert nach einem Beispiel aus „Operating Systems Internals and Design Principles“[Sta11, S. 237]).

der Monitor implizit freigegeben und erlaubt einem anderen Kontrollfluss den Eintritt in den Monitor [Hoa74, Sta11]. Signalisiert nun ein Kontrollfluss über die Operation *signal()*, dass eine bestimmte Bedingung eingetreten ist, so wird der erste Kontrollfluss aus der Warteschlange der betreffenden Variablen entnommen und seine Ausführung fortgesetzt [Hoa74, Sta11]. Da der vormals schlafende Kontrollfluss sofort seine Arbeit wieder aufnimmt und den Monitor betritt, verlässt der signalisierende Kontrollfluss unmittelbar den Monitor [Hoa74, Sta11]. Er wird in eine Vorzugswarteschlange eingefügt [Hoa74, Sta11]. Wird der Monitor nun wieder verfügbar, so wird zunächst ein Kontrollfluss aus der Verzugswarteschlange der Zugang gewährt [Hoa74, Sta11]. Erst wenn diese leer ist, werden neue Kontrollflüsse in den Monitor gelassen [Hoa74, Sta11].

2.2.1.5 *Message Passing*

Das Grundprinzip von *Message Passing* ist das Austauschen von Informationen zwischen Kommunikationspartnern über Nachrichten [Sta11]. Kommunikationspartner können sowohl mehrere Kontrollflüsse auf einem System als auch Kontrollflüsse auf unterschiedlichen Systemen sein [Sta11]. Die zur Kommunikation erforderlichen Operationen heißen: *send()* und *receive()* [Sta11]. Beide können blockierend und nicht-blockierend ausgelegt sein. Die gängigste Kombination ist das nicht-blockierende Senden und das blockierende Empfangen. Für das Versenden bzw. das Empfangen sind natürlich ein Absender und ein Adressat erforderlich. Bei der direkten Kommunikation werden diese bei der jeweiligen Operation direkt als Parameter übergeben [Sta11]. Sind die Teilnehmer der Kommunikation im Vorhinein jedoch nicht bekannt, wird die indirekte Kommunikation verwendet [Sta11]. Hier wird

ein Briefkasten[9] eingesetzt, der als Puffer für die Nachrichten dient [Sta11]. Dabei kann es sich z. B. um einen geteilten Speicher handeln [Sta11]. Alle Nachrichten werden nicht direkt an einen Empfänger versendet, sondern werden in einem Briefkasten zwischengespeichert [Sta11]. Aus diesem Briefkasten können Empfänger mit der Operationen *receive()* und *send()* Nachrichten abrufen bzw. senden. Entscheidend hierbei ist, dass eine Nachricht immer nur an einen Empfänger zugestellt wird. Evtl. weitere blockierte Empfänger warten solange, bis eine weitere Nachricht eintrifft.

Zur Realisierung von gegenseitigem Ausschluss wird eine indirekte Kommunikation mit einem nicht-blockierenden Senden und einem blockierenden Empfangen benötigt [Sta11]. Wie in Listing 2.3 in Zeile 1 zu sehen ist, wird ein Briefkasten initial mit einer Nachricht befüllt. Von nun an können alle N konkurrierende Kontrollflüsse über die zuvor genannten Operationen Nachrichten austauschen [Sta11]. Aufgrund der bereits erwähnten Semantik ist für die Kontrollflüsse in Listing 2.3 gegenseitiger Ausschluss gewährleistet [Sta11].

2.2.1.6 *Unterbrechungsbehandlungen*

Unterbrechungen sind ein grundlegender Mechanismus, der auf jeder Hardwarearchitektur zur Verfügung steht und in jedem Betriebssystem genutzt wird [Sta11, TB14b]. Es wird zwischen zum aktuellen Kontrollfluss synchronen und asynchronen Unterbrechungen unterschieden [Sta11, TB14b]. Letztgenannte sind für die folgenden Betrachtung von Interesse. Asynchrone Unterbrechungen werden von der Hardware ausgelöst [Sta11, TB14b]. Ein klassisches Beispiel für eine asynchrone Unterbrechung durch die Hardware ist die Signalisierung durch den Zeitgeberbaustein [Sta11, TB14b]. Dieser signalisiert dem Betriebssystem den Ablauf einer bestimmten Zeitspanne, nach der zu einem anderen Kontrollfluss gewechselt werden soll [Sta11, TB14b]. Die Behandlung dieser asynchronen Hardwareunterbrechung obliegt dem Betriebssystem [Sta11, TB14b]. Um die eigentliche Ausführung schnellstmöglich fortzuführen, kann das Betriebssystem die bei der Behandlung anfallenden Daten an eine nachgelagerte Unterbrechungsbehandlung in Software weiter reichen [SP94, Sta11, TB14b]. Dabei kann es mehrere Stufen einer nachgelagerten Unterbrechungsbehandlung mit absteigender Priorität geben [SP94, Lov10, Sta11, TB14b]. Möchten zwei oder mehr Stufen Daten miteinander austauschen, müssen sie sich eine geeignete Datenstruktur[10] teilen [Lov10, Sta11, TB14b]. Da die Stufen mit höherer Priorität die Ausführung von Stufen mit niedrigerer Priorität unterbrechen können, muss eine Synchronisation stattfinden [Lov10, Sta11, TB14b]. Hierzu werden die Stufen höherer Priorität deaktiviert [Lov10, Sta11, TB14b]. Jede Stufe, die einzeln deaktiviert werden kann, lässt sich daher auch als einen einzelnen Mechanismus zum gegenseitigen Ausschluss und somit als Sperre verstehen.

9 Deutsch für *mailbox*

10 Abseits von klassischen Datenstrukturen existieren auch sogenannte unterbrechungstransparente Datenstrukturen, die das Deaktivieren der Unterbrechungen überflüssig machen [SSSS00]. Solche Datenstrukturen spielen hier keine Rolle.

```
1  struct foo {
2      int a;
3  };
4  struct foo *gp = NULL;
5
6  void write_side(void) {
7      struct foo *p;
8      p = kmalloc(sizeof(*gp), GFP_KERNEL);
9      p->a = 1;
10     rcu_assign_pointer(gp, p);
11 }
12
13 void read_site(void) {
14     struct foo *p;
15     rcu_read_lock();
16     p = rcu_dereference(gp);
17     if (p != NULL) {
18         do_something_with(p->a);
19     }
20     rcu_read_unlock();
21 }
22
23 void update_side(void) {
24     struct foo *p, *q;
25     q = gp;
26     p = kmalloc(sizeof(*gp), GFP_KERNEL);
27     p->a = 42;
28     rcu_assign_pointer(gp, p);
29     synchronize_rcu();
30     kfree(q);
31 }
```

Listing 2.4: Ein Beispiel zur Synchronisation eines Leser-Schreiber-Problems mittels des Mechanismus' *Read-Copy-Update*. Die Funktionen `rcu_assign_pointer()` und `rcu_dereference()` stellen das atomare Lesen bzw. Schreiben des globalen Zeigers sicher. Es kann immer nur eine der beiden Funktionen `write_side()` und `update_side()` parallel zu der Funktion `read_side()` ausgeführt werden. (Das Beispiel ist aus einer Erklärung aus dem zugehörigen *LWN.net*-Artikel entnommen [Pau07a]. Mit Genehmigung von Paul McKenney ©2007.)

2.2.1.7 *Read-Copy-Update*

Der *Read-Copy-Update*-Mechanismus (kurz *RCU*) des Linux-Kerns dient der Synchronisation eines Leser-Schreiber-Problems, bei dem es viele gleichzeitige Leser und einen konkurrierenden Schreiber gibt [Pau07a]. Anders als Leser-Schreiber-Sperren herrscht hierbei kein gegenseitiger Ausschluss zwischen den Lesern und dem Schreiber [Pau07a]. Vielmehr sorgt der *RCU*-Machnismus mit den folgenden drei Prinzipien für eine Koexistenz von Lesern und einem Schreiber: 1) *Publish-Subscribe*-Mechanismus, 2) Einhalten einer *Grace*-Periode bis alle Leser ihre Arbeit beendet haben und 3) das Verwalten mehrerer Versionen von kürzlich aktualisierten Objekten [Pau07a]. Die drei Prinzipien lassen sich anhand des Beispiels in Listing 2.4 erklären: Das geteilte Datum ist der Zeiger `gp` auf eine Instanz der Datenstruktur `struct`

`foo` [Pau07a]. Die Allokation, Initialisierung und Zuweisung des Zeigers finden in der Funktion `write_side()` statt. Damit das Objekt nicht zu früh für Leser sichtbar ist, stellt die Funktion `rcu_assign_pointer()` in Zeile 10 sicher, dass die Instruktionen in der Funktion weder durch den Übersetzer noch den Prozessor umsortiert werden [Pau07a]. Hiermit wird das sog. Veröffentlichen[11] realisiert [Pau07a]. Das Gegenstück findet sich in Zeile 16 in der parallel dazu ausgeführten Funktion `read_side()`. Die Funktion `rcu_dereference()` bildet das Abonnieren[12] ab [Pau07a]. Durch die Aufrufe in Zeile 15 bzw. 20 teilt ein Leser sein Interesse an einer mittels *RCU* abgesicherten Datenstruktur mit [Pau07a]. Ist nun eine Aktualisierung der Datenstruktur erforderlich, wird anstatt der Funktion `write_side()` die Funktion `update_side()` ausgeführt. Hier wird eine neue Instanz alloziert und initialisiert. Die Zuweisung des Zeigers findet erneut mittels der Funktion `rcu_assign_pointer()` statt [Pau07a]. Nach der Ausführung von Zeile 28 werden zukünftige Leser nur noch auf der neuen Instanz arbeiten können [Pau07a]. Doch bevor die alte Instanz in Zeile 30 freigegeben werden kann, muss mit Hilfe der Funktion `synchronize_rcu()` in Zeile 29 gewartet werden, bis alle existierenden Leser ihren kritischen Abschnitt beendet haben [Pau07a]. Auf diese Weise werden mehreren Versionen von Objekten verwaltet und nach Ablauf einer *Grace*-Periode werden die alten Objekte freigegeben [Pau07a]. Sollte es entgegen des zuvor geschilderten Szenarios dennoch erforderlich sein, mehrere konkurrierende, schreibende Kontrollflüsse zu haben, so müssen diese sich weiterhin untereinander mit einer gewöhnlichen Sperre synchronisieren [Pau07a]. Mit den vorgenannten Prinzipien kann der *RCU*-Mechanismus u.a. als eine Leser-Schreiber-Sperre oder als ein einfacher Referenzzähler eingesetzt werden [Pau07b]. Die vorgenannte Erläuterung schildert die für das Verständnis wesentlichen Grundprinzipien. Die Programmierschnittstelle umfasst jedoch mehr Funktionen [Pau08]. Ebenso sind die Implementierungen des *RCU*-Mechanismus ausgereifter geworden [Pau08].

2.2.1.8 *Sequential Locks*

Der Sperren-Typ *Sequential Locks* dient ebenfalls der Synchronisation von Leser-Schreiber-Problemen [Lov10]. Er ist für Szenarien mit vielen Lesern und wenigen Schreibern gedacht [Lov10]. Der Sperren-Typ besteht aus einem gewöhnlichen Spinlock und einem Zähler [Lov10]. Letzterer wird eingesetzt, um seitens der Leser einen gleichzeitigen Schreibvorgang zu detektieren [Lov10]. Vor dem Lesen der Daten wird der Zähler ausgelesen und zwischengespeichert [Lov10]. Nachdem die Daten gelesen wurden, wird der zuvor gespeicherte Zählerwert mit dem aktuellen Zähler verglichen [Lov10]. Sind sie nicht gleich, wurde der Lesevorgang von einem Schreibvorgang unterbrochen und muss nun wiederholt werden [Lov10]. Andernfalls muss der Lesevorgang nicht wiederholt werden [Lov10]. Das Schreiben wird zum einen mit dem Spinlock gegen andere, konkurrierende Schreiber abgesichert, zum anderen wird vor dem Schreibvorgang der Zähler erhöht, ebenso wie

11 Deutsch für *publish*.
12 Deutsch für *subscribe*.

nach dem Schreibvorgang [Lov10]. Somit kann sogar ein gerade in Arbeit befindlicher Schreibvorgang detektiert werden [Lov10]. Mit Hilfe des Typs *Sequential Lock* können schreibende Kontrollflüsse ihren Zugriff immer abschließen und werden von Lesern nicht verzögert [Lov10].

2.2.2 *Umgang mit Nebenläufigkeit*

In diesem Abschnitt wird ein Überblick über die Möglichkeiten an fehlerhaftem Umgang mit Nebenläufigkeit gegeben. Zwei Beispiele für fatale bzw. tödliche Konsequenzen wurden bereits eingangs dieser Arbeit in Kapitel 1 aufgezeigt. Diesen Beispiele ist gemein, dass die Annahmen, die die Entwickler über ihre Software bezüglich der Nebenläufigkeit machten, offenkundig nicht stimmen. Andernfalls wäre es nicht zu den geschilderten Ereignissen gekommen. Verallgemeinert ausdrückt: Die jeweilige Software arbeitete nicht korrekt.

Umgangssprachlich bedeutet die Korrektheit einer Software nach Ian Sommerville: „(...) ensuring the system services are as specified (...)“ ([Som16], S. 289) bzw. „(...) the correctness of a system can be defined by specifying how system inputs map to corresponding outputs that should be produced by the system.“ ([Som16], S. 612). Präziser lässt sich die Korrektheit beispielsweise in Form der Eigenschaften *safety* [Lam89] und *liveness* [Lam89] formulieren [Bus90]: Unter dem Aspekt *safety* kann man beispielsweise die Eigenschaft formulieren, dass nie mehr als ein Prozess einen kritischen Abschnitt betreten darf [Bus90]. Unter der Eigenschaft *liveness* kann z. B. der garantierte Fortschritt eines Kontrollflusses modelliert werden [Bus90].

Unabhängig davon, wie die Korrektheitsanforderungen formuliert werden, können verschiedene Fehler diese gefährden: Dies war bereits Gegenstand verschiedener Arbeiten[13]: Lu et al. [LPSZ08] sowie Asadollah et al. [AASEH17] untersuchten verschiedene quelloffene Softwareprojekte hinsichtlich der Diversität von Programmfehlern, die im Zusammenhang mit Nebenläufigkeit stehen. Ihre Resultate lassen sich auf zwei Arten von Nebenläufigkeit zurückführen: a) Bei der versteckten Nebenläufigkeit ist sich der Entwickler einer möglichen Nebenläufigkeit nicht bewusst. Es kommen keinerlei Synchronisationsmittel zum Einsatz. b) Bei der bewussten Nebenläufigkeit hingegen sichert der Entwickler seinen Programmcode gegen mögliche konkurrierende Kontrollflüsse ab. Allerdings treten Fehler bei der Implementierung, wie z. B. Verklemmungen, auf. Beim Umgang mit bewusster Nebenläufigkeit treten die Fehler bei der Realisierung der Synchronisation auf. Diese lassen sich ebenfalls in Kategorien unterteilen, die im Folgenden noch vorgestellt werden.

Zur Verdeutlichung was unter unbewusster Nebenläufigkeit zu verstehen ist, sollen einzelne Resultate aus der Arbeit von Lu et al. vorgestellt werden: Sie schauten sich dazu vier quelloffene Softwareprojekte an und wählten 105 zufällige Programmierfehler, die in Zusammenhang mit Nebenläufigkeit

13 Arpaci et al. greifen ebenfalls die Arbeit von Lu et al. auf und führen anschließend eine ähnliche Unterteilung durch [ADAD18].

```
1  void kontrollfluss1() {
2      // ...
3      if (thd->proc_info) {
4          fputs(thd->proc_info, ...);
5      }
6      // ...
7  }
8
9  void kontrollfluss2() {
10     // ...
11     thd->proc_info = NULL;
12     // ...
13 }
```

Listing 2.5: Ein exemplarischer Programmierfehler aus dem *MySQL*-Projekt aus der Datei *ha_innodb.cc*. Hierbei wird die Annahme der Atomarität im Programmcode von Kontrollfluss 1 in den Zeilen 3 und 4 verletzt [LPSZ08]. (Adaptiert und entnommen aus einer Abb. von Lu et al. [LPSZ08])

stehen, aus [LPSZ08]. Davon haben 74 Fehler nichts mit Verklemmungen zu tun, 31 Fehler hingegen schon [LPSZ08]. Die Erstgenannten teilen sie weiter in drei Kategorien auf [LPSZ08]. Zwei davon sollen nun als Beispiele für unbewusste Nebenläufigkeit vorgestellt werden. Nach ihren Untersuchungen fallen 72 von 74 Fehlern in diese beiden Kategorien [LPSZ08].

ATOMICITY VIOLATION Bei der Verletzung der Atomarität wird die Annahme verletzt, dass die Speicherzugriffe eines bestimmten Abschnitts im Programmcode atomar ablaufen [LPSZ08]. Gleichzeitig werden keinerlei Hilfsmittel genutzt, um die Atomarität zu erzwingen [LPSZ08]. Mit dem Beispiel in Listing 2.5 aus dem *MySQL*-Projekt[14] zeigen Lu et al., wie dies ablaufen kann [LPSZ08]. In Kontrollfluss 1 wird implizit angenommen, dass die dortigen Anweisungen in Zeile 3 und 4 atomar ausgeführt werden [LPSZ08]. Hierdurch soll sichergestellt werden, dass dem Funktionsaufruf von `fputs()` eine gültige Adresse übergeben wird. Da eine atomare Ausführung jedoch nicht zwingend gegeben ist, kann die Anweisung aus Kontrollfluss 2 in Zeile 11 verschachtelt dazu ausgeführt werden. Sie setzt die in `thd->proc_info` abgelegte Adresse zurück und sorgt somit dafür, dass dem Funktionsaufruf eine ungültige Adresse übergeben wird.

ORDER VIOLATION Eine Verletzung der Reihenfolge tritt auf, wenn zwei Gruppen von Speicherzugriffen nicht in der vom Programmierer intendierten Reihenfolge ausgeführt werden [LPSZ08]. Auch hier wird die vorgesehene Reihenfolge nicht durch Hilfemittel forciert [LPSZ08]. In Listing 2.6 ist zu sehen, wie zwei konkurrierende Kontrollflüsse auf den Zeiger `mThread` zugreifen. Dabei initialisiert Kontrollfluss 1 ihn, während Kontrollfluss 2 den Zeiger dereferenziert. Der Programmierer nahm an, dass Zeile 10 erst nach

14 https://www.mysql.com

```
1  // Kontrollfluss 1
2  void init() {
3      // ...
4      mThread = PR_CreateThread(mMain, ...);
5      // ...
6  }
7
8  // Kontrollfluss 2
9  void mMain() {
10     mState = mThread->State;
11     // ...
12 }
```

Listing 2.6: Ein exemplarischer Programmierfehler aus dem *Mozilla*-Projekt. Der Programmierer nahm an, Zeile 10 wird durch Kontrollfluss 2 nach Kontrollfluss 1 ausgeführt und somit `mThread->state` erst gelesen, nachdem `mThread` in Zeile 4 beschrieben wurde [LPSZ08]. (Adaptiert und entnommen aus einer Abb. von Lu et al. [LPSZ08])

dem Beschreiben des Zeigers in Zeile 4 ausgeführt wird. Tatsächlich ist aber auch die eine andere Ausführungsreihenfolge möglich: Es wird zuerst Zeile 10 ausgeführt [LPSZ08]. Letztgenannte Ausführungsreihenfolge führt letztlich zu einer Speicherzugriffsverletzung [LPSZ08]. Neben diesem Beispiel zeigen Lu et al. in ihrer Arbeit noch weitere, komplexere Beispiele für die Verletzung der Reihenfolge von Speicherzugriffen auf [LPSZ08].

2.2.2.1 *Fairness*

Unter Fairness ist die faire Zuteilung eines kritischen Abschnitts unter den konkurrierenden Kontrollflüssen zu verstehen. Ist ein kritischer Abschnitt bereits durch einen Kontrollfluss belegt, müssen alle weiteren Kontrollflüsse warten. Unabhängig davon, ob sie passiv oder aktiv warten, gilt es beim Freiwerden des kritischen Abschnitts, als nächstes einem Kontrollfluss den Zugang zu gewähren. Im Folgenden wird die Fairness bei der Nutzung von Synchronisationsprimitiven, die aktives und passives Warten verwenden, eingegangen.

Stellvertretend für Synchronisationsprimitiven, die auf passivem Warten aufbauen, soll auf die Fairness von Semaphore eingegangen werden. Da beim passivem Warten der betreffende Kontrollfluss den Prozessor abgibt, muss er in einer Datenstruktur abgelegt werden, um ihn entsprechend wiederzufinden, wenn der kritische Abschnitt wieder verfügbar ist [Sta11]. Dafür wird traditionell eine Warteschlange verwendet [Sta11]. Die Fairness ergibt sich aus der Reihenfolge, in der die Kontrollflüsse entfernt werden [Sta11]. In der Literatur finden sich nun unterschiedliche Einordnungen der Auswahlstrategien [Sta82, Pet83, Sta11, HI13]. Für das grundlegende Verständnis genügt die Sichtweise aus dem Buch „Operating Systems Internals and Design Priniciples“ [Sta11]. Es wird zwischen zwei Typen unterschieden: Die schwachen Semaphore[15] geben keinerlei Reihenfolge vor, in der

15 Deutsch für *weak semaphore* [Sta11], oder auch *blocked-set* genannt [Sta82]

Kontrollflüsse aus der Warteschlange entnommen werden [Sta11]. Bei den starken Semaphore[16] werden die Kontrollflüsse nach dem Prinzip *First-In-First-Out* aus der Warteschlange entnommen [Sta11]. Hierbei ist sichergestellt, dass der Kontrollfluss, der am längsten wartete, als Nächster an die Reihe kommt [Sta11].

Bei Synchronisationsprimitiven, bei denen aktiv gewartet wird, wird die Reihenfolge naturgemäß nicht vorweg genommen: In ihrer einfachsten Form wird bei Spinlocks fortwährend überprüft, ob der kritische Abschnitt frei ist und gleichzeitig versucht, diesen zu betreten (vgl. Unterabschnitt 2.2.1.3). Dem Kontrollfluss, dem dies zuerst gelingt, er hält Zugang. Dabei ist keinerlei Fairness garantiert [ADAD18]. Um diesen Missstand zu begegnen wurden sowohl in der Forschung [And90, MCS91] als auch in echten Betriebssystemen, wie z. B. Linux [Jon14a], Alternativen erarbeitet, die eine Reihenfolge vorgeben.

Unabhängig von der eingesetzten Synchronisationsprimitive wirkt sich die gewählte Fairness-Strategie auf die Latenz einzelner Kontrollflüsse aus. Wird ein einzelner Kontrollfluss oder eine Gruppe bevorzugt, so erhöht sich offensichtlich die Latenz der verbleibenden, konkurrierenden Kontrollflüssen [And90]. Im schlimmsten Fall führt dies zum Aushungern[17] von einzelnen Kontrollflüssen [SGG10, Sta11, TB14b].

2.2.2.2 *Skalierbarkeit*

Bei nebenläufiger Software ist die Intention, die anfallende Arbeitslast durch nebenläufige Kontrollflüsse erledigen zu lassen, um letztlich eine bessere Performanz zu erzielen. Bei einem Teil dieser Software, wie z. B. der Linux-Kern oder Datenbanksoftware [JPH+09], ist es erforderlich, Daten in Form einer geteilten Datenstruktur zwischen den Kontrollflüssen zu teilen und die Zugriffe darauf zu synchronisieren. Bei der Umsetzung der Synchronisation muss immer zwischen der Granularität des Sperrens oder Skalierbarkeit abgewogen werden, wie z. B. in Datenmanagementsystemen [JPH+09]. Zur Veranschaulichung werden daher im Folgenden verschiedene Kombinationen aus Betriebssystemkernen und Benchmarks qualitativ bezüglich der Kriterien Granularität des Sperrens sowie Grad der Parallelität eingeordnet und in Abbildung 2.2 zusammengefasst.

Wie in Abbildung 2.2 auf der Abszisse dargestellt ist, kann hier qualitativ zwischen *grob* und *fein* unterschieden werden. Wobei ersteres eine Sperre für alle geteilten Daten bezeichnet. Unter feingranularem Sperren, auf der anderen Seite, ist zu verstehen, wenn im Extremfall für jedes Element einer Datenstruktur eine separate Sperre verwendet wird. Die dazwischenliegenden Stufen hängen indes stark von dem konkreten Anwendungsfall ab und spielen für diese Betrachtungen eine untergeordnete Rolle. Ebenso wird hier der allgemeine Fall betrachtet, in dem immer nur eine Teilmenge der einer Datenstruktur zugeordneten Sperren für ein Element daraus nötig sind. An-

16 Deutsch für *strong semaphore* [Sta11], oder auch *blocked-queue* genannt [Sta82]
17 Deutsch für *starvation*

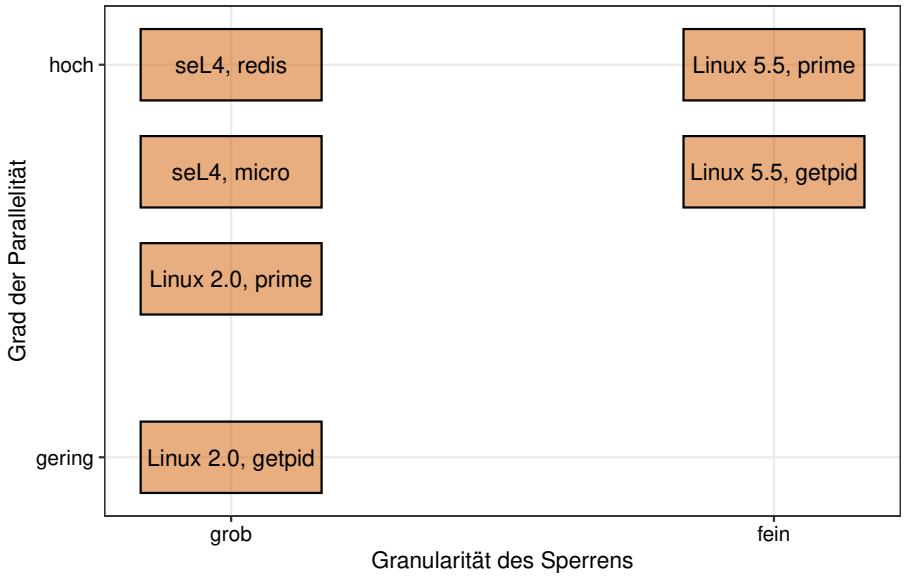


Abbildung 2.2: Qualitative Einordnung des Zusammenhangs zwischen dem Grad der Nebenläufigkeit und der Granularität des Sperrens in nebenläufiger Software. In einem organgenem Kasten wird der Betriebssystemkern sowie der verwendete Benchmark angegeben.

dernfalls lässt sich immer ein Fall konstruieren, in dem alle Sperren belegt werden müssen. Dieser Fall kommt dem Ein-Sperren-Fall gleich.

Die Ordinate stellt – ebenfalls qualitativ – den Grad der Nebenläufigkeit dar. Je weiter oben sich ein Datenpunkt befindet, desto mehr Kontrollflüsse verrichten gleichzeitig ihre Arbeit. Zusätzlich soll der Einfachheit halber zunächst angenommen werden, dass das Belegen bzw. Freigeben einer Sperre keine Zeit kostet sowie für die Sperre selbst kein Speicher benötigt wird.

Grundsätzlich gilt, dass ein feingranulareres Sperren von Datenstrukturen einen höheren Grad der Nebenläufigkeit erlaubt [CH93, CB08]. Peters et al. zeigten mit ihrer Arbeit jedoch, dass die Häufigkeit, mit der Sperren geholt werden, ebenfalls einen Einfluss haben [PDEH15]. Sie untersuchten den Einfluss von grobgranularem Sperren in Abhängigkeit von der Arbeitslast auf das Mikrokernbetriebssystem *seL4* [PDEH15]. Dabei zeigten sie mit Hilfe eines Mikrobenchmarks, dass bei einem hochfrequentem Zugriff auf einen schützenswerten Bereich im Betriebssystem grobgranulares Sperren der Nebenläufigkeit nicht zuträglich ist [PDEH15]. Dies wird durch den Datenpunkt *(seL4,micro)* in Abbildung 2.2 symbolisiert. Wird im Gegensatz dazu eine realistische Arbeitslast[18] gewählt und die Abstände zwischen dem Zugriff auf die geschützten Daten sind groß, so genügt eine Sperre um den Mikrokern abzusichern, aber dennoch eine entsprechende Nebenläufigkeit zu erzielen [PDEH15] – durch den Datenpunkt *(seL4, redis)* dargestellt.

Nach dem gleichen Schema lässt sich auch der Linux-Kern einordnen: Mit der Unterstützung für symmetrische Multiprozessorsysteme in Linux 2.0 [Lov10] wurde das sog. *Big Kernel Lock* eingeführt [Lov10, Jon14a]. Hier sichert eine Sperre die Datenstrukturen im Betriebssystemkern ab [Lov10,

18 Die Autoren nutzten die Software *redis* (http://redis.io).

Jon14a]. Dies hat entsprechend zur Folge, dass immer nur ein Kontrollfluss den Kern betreten kann [Lov10, Jon14a]. Unter Berücksichtigung der bereits vorgestellten Arbeiten [CH93, CB08, PDEH15] lassen sich die folgenden zwei Benchmarks ebenfalls qualitativ einordnen: Läuft auf einem Linux-System ein paralleler Benchmark mit wenig Interaktion mit dem Betriebssystemkern, wie z. B. die Berechnung von Primzahlen, so ist dennoch ein angemessener Grad an Parallelität zu erwarten – vgl. *(Linux 2.0, prime)* in Abbildung 2.2. Anders sieht dies hingegen bei einem Benchmark aus, der viel Kerninteraktion hat, wie z. B. ein Programm, das parallel immer wieder den Systemaufruf *getpid()*[19] aufruft. Aufgrund des grobgranularen Sperrens würde hier wenig nebenläufige Aktivität stattfinden – vgl. *(Linux 2.0, getpid)*. Da der Linux-Kern mittlerweile verschiedene Mechanismen zum feingranularen Sperren bereitstellt [Lov10], konnte der erreichbare Grad der Nebenläufigkeit erhöht werden. Daher finden sich in Abbildung 2.2 zum Vergleich noch zwei Datenpunkte für eine aktuelle Version[20] des Linux-Kerns mit den zwei entsprechenden Benchmarks.

Neben den oben genannten Beziehungen kommt in der Realität noch hinzu, dass eine Sperre Speicherplatz belegt und die darauf ausgeführten Operationen Zeit benötigen. Daher ist in der Praxis der Abbildung 2.2 noch eine weitere Dimension hinzuzufügen. Sie lässt sich unter dem Begriff Effizienz hinsichtlich Speicher und Laufzeit zusammenfassen: Je mehr Sperren für eine Datenstruktur verwendet werden, desto mehr Speicherplatz wird benötigt.

Des Weiteren spielt in der Praxis die Wartbarkeit der Software eine Rolle. Sie verschlechtert sich desto mehr Sperren involviert sind, da sie die Komplexität erhöht [CB08].

2.2.2.3 *Korrektes Sperren*

Zur Sicherung eines kritischen Abschnitts gegen konkurrierende Zugriffe muss mit Hilfe von Synchronisationsprimitiven gegenseitiger Ausschluss hergestellt werden. Wenngleich es hierzu eine Vielzahl von Möglichkeiten gibt, wie Unterabschnitt 2.2.1 bereits zeigte, wird sich für die folgenden Betrachtungen auf explizite Synchronisationsmittel, wie Semaphore oder Spinlocks, beschränkt. Bei der Verwendung von einer oder mehrerer Sperren kann es zu zweierlei Problemen kommen: a) Es werden weniger Sperren als vorgesehen eingesetzt. b) Es werden mehr Sperren als vorgesehen eingesetzt.

Werden nicht alle vorgesehenen Sperren geholt, ist kein gegenseitiger Ausschluss gewährleistet und es besteht die Gefahr einer Wettlaufsituation [Sta11]. Es besteht die Gefahr der Korruption der Daten [Sta11]. Im besten Fall bleibt dies ohne Folgen. Im schlimmsten Fall führt dies jedoch zu einer

19 Der Systemaufruf *getpid()* führt selbst keine Arbeit aus, außer die Prozess-ID des Aufrufers zurückzugeben [Ous90]. Er wird zur Messung der Kosten von Kernein- und austritt verwendet [Ous90].

20 Zum Zeitpunkt der Entstehung dieser Arbeit war dies die Verion 5.5 des Linux-Kerns.

fatalen Fehlfunktion des Betriebssystems oder der Software im Allgemeinen wie das Beispiel des Therac-25 zeigte – siehe Kapitel 1.

Das Verwenden von mehr Sperren als eigentlich nötig sind, hat keinen Einfluss auf die Korrektheit des Systems[21]. Allerdings schränkt solch ein pessimistisches Vorgehen den Grad der Parallelität von Software ein ein – siehe Unterabschnitt 2.2.2.2 [LSBS19].

Orthogonal zu den beiden vorgenannten Problemen ist der Umgang mit der Reihenfolge, in der die jeweiligen Sperren geholt werden: Fordern mindestens zwei konkurrierende Kontrollflüsse gleichzeitig zwei oder mehr Sperren in unterschiedlicher Reihenfolge an, besteht die Gefahr einer Verklemmung[22] [SGG10, Sta11, TB14b]. Hierbei blockieren zwei (oder mehr) Kontrollflüsse permanent, da sie wechselseitig aufeinander warten, ohne dass auch nur ein Kontrollfluss irgendeinen Fortschritt erfährt[23] [SGG10, Sta11, TB14b]. Es ist eine Verklemmung eingetreten. Das Szenario lässt sich zu den vier folgenden Bedingungen für Verklemmungen verallgemeinern [SGG10, Sta11, TB14b]:

1. *Mutual Exclusion*: Eine Ressource kann zu einem bestimmten Zeitpunkt nur von einem Kontrollfluss belegt werden [SGG10, Sta11, TB14b].

2. *Hold and wait*: Während ein Kontrollfluss auf die Zuweisung einer weiteren Ressource wartet, gibt er die bereits erhaltenen Ressourcen nicht wieder her [SGG10, Sta11, TB14b].

3. *No preemption*: Eine einmal einem Kontrollfluss zugeteilte Ressource kann ihm nicht mehr entzogen werden [SGG10, Sta11, TB14b].

4. *Circular Wait*: Es existiert eine geschlossene Kette an wechselseitig auf einander wartenden Kontrollflüssen [SGG10, Sta11, TB14b]. Dabei wartet jeder Kontrollfluss auf eine Ressource, die von dem nächsten Kontrollfluss in der Ketter belegt ist [SGG10, Sta11, TB14b].

Da die Definition einer Verklemmung allgemein verfasst wurde, ist entsprechend von Ressourcen die Rede. In dem vorliegenden Fall entsprechen die eingesetzten Sperren den Ressourcen. Die Möglichkeit von Verklemmungen ist gegeben, wenn die ersten drei Bedingungen zutreffen [SGG10, Sta11, TB14b]. Tritt auch die vierte Bedingungen ein, ist eine Verklemmung unausweichlich [SGG10, Sta11, TB14b].

21 Die Betrachtung der Reihenfolge wird hier noch ausgeklammert.

22 Deutsch für *Deadlock*

23 Kommen Sperren zum Einsatz, bei denen aktiv gewartet wird, wie z. B. ein Spinlock, spricht man von einem *Livelock*.

3 PROBLEMBESCHREIBUNG

*"Locking is a necessary evil in operating systems;
without a solid locking regime, different parts of the system will collide
when trying to access the same resources,
leading to data corruption and general chaos."*

— Jonathan Corbet, Linux-Kern-Entwickler

Inhalt

Wie eingangs in Kapitel 1 bereits erwähnt wurde, geht der Trend bei heutigen Prozessoren hin zu Multi- und Vielkern-Systemen. Dies führte bei monolithischen Betriebssystemen, wie Linux, zu feingranularerem Sperren innerhalb des Betriebssystemkerns, so dass ein höherer Grad an Parallelität möglich wurde (vgl. Unterabschnitt 2.2.2.2 sowie Unterabschnitt 2.1.1). Feingranulareres Sperren erhöht implizit die Komplexität, da es mehrere Sperren gibt, die jeweils einem unterschiedlichen Zweck dienen. Dieser sollte, ähnlich wie Quellcode, kontinuierlich für die Entwicklergemeinde dokumentiert werden.

Im Folgenden wird anhand von Linux und FreeBSD [MNNW14] erläutert, wie sich solch eine Dokumentation von feingranularerem Sperren von existierenden Betriebssystemen dem Entwickler präsentiert. Dazu zählt, welche Form die Dokumentation aufweist, wo sie zu finden ist und wie aktuell sie ist. Die sog. Sperren-Dokumentation umfasst einerseits Informationen über die Sperren-Typen und andererseits den Zweck, dem die bereits im Betriebssystemkern vorhandenen Sperren dienen. Zudem sollte sie an einer zentralen Stelle zu finden sein und stets aktuell gehalten werden, damit sie einen Beitrag zu der Stabilität eines Betriebssystemkerns leisten kann. Sie sollte ebenfalls präzise formuliert sein, damit dem Entwickler stets klar ist, welche Sperren zu verwenden sind. Andernfalls steigt das Risiko, dass ein Entwickler Fehler macht.

Aus den geschilderten Gegebenheiten der einzelnen Betriebssystemen lassen sich zum Abschluss die für diese Arbeit relevanten Problemstellungen ableiten.

3.1 LINUX

Im Fall von Linux führte der oben beschriebene Weg über die letzten Jahre von einer einzigen Sperre, dem sog. *Big Kernel Lock* [Lov10, Jon14b], hin

zu einer Vielfalt an Sperr-Mechanismen, die z. B. Teilbereiche des Kerns oder Instanzen von Datenstrukturen schützen. Natürlich dienen diese Mechanismen nicht nur zur Multiprozessorsynchronisation, sondern können auch zur Synchronisation von anderen Arten der Nebenläufigkeit, wie z. B. Multiprozess- oder Unterbrechungssynchronisation, genutzt werden [Lov10]. Abbildung 1.1 stellt die Nutzung[1] der zwei Sperren-Typen *mutex* und *spinlock* über die letzten 10 Jahre dar. Außerdem zeigt es die Entwicklung der Codegröße über denselben Zeitraum. Diese nahm um 110,4 % zu. Die Nutzung von Sperren des Typs *mutex* bzw. *spinlock* stiegen beispielsweise um 111,24 % sowie 54,23 % an. Eine ähnliche Entwicklung zeigt die Anzahl an Kommentaren, die sich mit Synchronisation beschäftigen. Hierzu wurden die Quellcodedateien jeder Version des Kerns nach verschiedenen Stichworten[2] durchsucht. Wie in Abbildung 1.1 zu sehen ist, steigt die Anzahl an Kommentaren um 91,55 % an. Die Zahlen sind ein Indiz dafür, dass auch neuer Code genauso viel mit Sperren arbeitet wie existierender Code.

Somit kommt der korrekten Verwendung von Sperren eine steigende Bedeutung bei, da der falsche Umgang mit Sperren sonst Auswirkungen auf das gesamte Betriebssystem haben könnten, wie Unterabschnitt 2.2.2.3 bereits zeigte. Die Literatur bietet einen Überblick über die verschiedenen Mechanismen, die der Linux-Kern bereitstellt [BC05, Lov10, Rus]. Hier wird bereits die Komplexität des Problems deutlich: Im Linux-Kern existieren in der Größenordnung acht Sperren-Typen[3] [Lov10]. Hinzu kommt, dass in Abhängigkeit von der Kombination aus aufrufendem Kontext und konkurrierenden Kontext sichergestellt muss, dass der aufrufende Kontext den konkurrierenden Kontext aussperrt [LSBS19]. Dies ist beispielsweise gegeben, wenn der aufrufende Kontext – ein Thread – vor dem Belegen der Sperre die Unterbrechungen sperrt. So verhindert er, dass ein konkurrierender Kontext aus einer Unterbrechungsbehandlung (auf dem demselben Prozessor) heraus nicht für eine Verklemmung sorgt [LSBS19].

Neben den vorgenannten Möglichkeiten des Sperrens bietet der Linux-Kern auch sogenannte *lock-free* Datenstrukturen, die gänzlich ohne Sperren auskommen, und sowie Operationen für den atomaren Zugriff an [BC05, Lov10, Rus].

In der Literatur werden diese Möglichkeiten und Zusammenhänge zwar erläutert, allerdings werden nur ganz allgemein die korrekte Verwendung sowie die möglichen Gefahren geschildert [BC05, Lov10, Rus]. Es ist für einen Entwickler jedoch ebenso wichtig, welche konkrete Sperre einen bestimmten Speicherbereich, z. B. eine Instanz einer Datenstruktur, schützt. Die sogenannten Sperren-Regeln schaffen eine Zuordnung von zu schützenden Daten zu einer oder mehreren Sperren. Nur mit dieser Information ist es einem

1 Es wurde gezählt, wie häufig die jeweiligen Initialisierungsfunktionen der betreffenden Sperren-Typen aufgerufen wurden.

2 Regulärer Ausdruck zur Suche nach synchronisations-relevevanten Quellcodekommentaren: `<racing>|<race>|protected by|caller holds|protects|should be holding|called under|must be held|lock ordering|lock <order>`.

3 Die genaue Anzahl hängt letztlich von der Kategorisierung ab.

```
/*
 * Inode locking rules:
 *
 * inode->i_lock protects:
 *   inode->i_state, inode->i_hash, __iget()
 * Inode LRU list locks protect:
 *   inode->i_sb->s_inode_lru, inode->i_lru
 * [...]
 * inode_hash_lock protects:
 *   inode_hashtable, inode->i_hash
 * [...]
 */
```

Listing 3.1: Ein Auszug der Sperren-Dokumentation aus der Datei `linux-v5.6/fs/inode.c` für die Datenstruktur `struct inode`. (Nach einer Vorlage von Lochmann et al. [LSBS19])

Entwickler möglich, seinen Programmcode gegen konkurrierende Zugriffe zu schützen und so seine Modifikation fehlerfrei umzusetzen [LSBS19].

Im Fall von Linux steht dieser Information jedoch nicht an einer zentralen Stelle im Quellcode [LSBS19]. Vielmehr ist sie über den gesamten Quellcode verteilt [LSBS19]. Eine erste Anlaufstelle ist typischerweise die Stelle im Programmcode, wo eine Datenstruktur definiert wird [LSBS19]. Die Definition der Datenstruktur `struct inode`[4] beispielsweise enthält lediglich einen Quellcodekommentar, der erahnen lässt, worum es sich handeln kann:

```
spinlock_t i_lock; /* i_blocks, i_bytes, maybe i_size */
```

[LSBS19]. Abgesehen davon gibt es lediglich eine weitere zentrale Stelle, wo weitere Informationen über Sperren-Regeln für `struct inode` stehen: Ein Auszug der Datei `fs/inode.c`, die eine der zentralen Quellcodedateien im Zusammenhang mit der Struktur `inode` darstellt, ist in Listing 3.1 dargestellt [LSBS19].

Nach derzeitigem Stand und besten Wissen des Autors handelt es sich hierbei um die einzigen beiden zentralen Stellen im Linux-Kern, die einem Entwickler Informationen über das korrekte Sperren der Datenstruktur `struct inode` geben [LSBS19]. Dennoch enthält der restliche Quelltext des Linux-Kerns hierzu weitere Dokumentation [LSBS19]. Diese ist jedoch über verschiedene *C*-Dateien in den Verzeichnissen `fs/` sowie `include/` verteilt [LSBS19]. Neben der im Quellcode eingebetteten Informationen stellt der Linux-Kern im Verzeichnis `Documentation/` selbst Dokumentation bereit [LSBS19]. Dies beinhaltet sogar Sperren-Regeln[5] [LSBS19]. Da diese jedoch speziell auf Anwendungsfälle, wie z. B. das Öffnen einer Datei, ausgerichtet sind, lassen sich nur schwer Rückschlüsse auf das korrekte Sperren bei einem Zugriff auf einzelne Elemente einer Datenstruktur ziehen [LSBS19].

4 Definiert in `linux-v5.6/include/linux/fs.h`.

5 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/locking.rst?h=v5.6
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/directory-locking.rst?h=v5.6

Neben der Vielzahl von Stellen, wo Dokumentation zu finden ist, zeigt sich bei der genauen Untersuchung der einzelnen Sperren-Regeln ein weiteres Problem: Die Formulierungen sind nicht eindeutig [LSBS19]. Stichproben aus Quellcodedateien des Datei-Subystems in `fs/*.c` zeigen, dass unterschiedliche Ausdrücke verwendet werden, um dem Entwickler mitzuteilen, dass eine bestimme Sperre gehalten werden muss [LSBS19]. Dies sind beispielsweise „holds“ oder „is held“ [LSBS19]. Dieses Muster setzt sich bei den Namen der Sperren fort: Anstatt den exakten Namen der Variable zu verwenden, werden öfters auch Beschreibungen einer Sperre verwendet [LSBS19]. Dies lässt sich anhand der Funktion `find_inode()`, die eine Liste von Inodes mit Hilfe des Elements `i_hash` durchsucht, zeigen [LSBS19]. Gemäß des Quellcodekommentars zu Beginn der Funktion soll diese nur betreten werden, wenn das sog. „inode lock“ gehalten wird: „Called with the inode lock held.“[6] [LSBS19]. Betrachtet man jedoch, aus welcher Funktion die Funktion `find_inode()` aufgerufen wird, so stellt sich heraus, dass zusätzlich noch die Sperre `inode_hash_lock` gehalten wird [LSBS19]. Ein Blick in die Dokumentation in Listing 3.1 teilt dem Entwickler in den Zeilen 4, 6 und 9 mit, beide Sperren zu verwenden. Dies steht zumindest zum Teil in einem Widerspruch zu den Kommentaren im Quellcode [LSBS19]. Unterm Strich wird dem Entwickler durch bloßes Lesen der Dokumentation nicht eindeutig klar, welche Sperre(n) genau zu verwenden sind [LSBS19].

Sowohl die inkonsistente Benennung von Sperren als auch die unklare Formulierung der Regeln machen es für den Entwickler undurchsichtig, welche Sperren gemeint sind [LSBS19]. Außerdem erschwert es eine automatische Extraktion der Sperren-Regeln aus dem Quelltext, was aber wünschenswert ist, um den Quellcode automatisch auf Fehler zu überprüfen.

Darüber hinaus gibt es Anzeichen, dass die Entwickler des Linux-Kerns sich selbst nicht sicher sind, wie genau zu sperren ist. In Listing 3.2 ist der Kommentar zu der Funktion `inode_set_flags()` dargestellt.

Hier wird in Zeile 6 ff. erläutert, warum die Implementierung unterbrechungstransparent ausgelegt ist, obwohl eigentlich eine Sperre genügt: Nicht alle Codepfade halten sich an das Abkommen die vorgesehene Sperre zu verwenden [LSBS19]. Es sei noch angemerkt, dass die in dem Text erwähnte Sperre `i_mutex` in der aktuellen Version des Linux-Kerns – v5.6 – nicht mehr existiert.

An einer weiteren Stelle im Quellcode des Linux-Kerns heißt es: *„We don’t actually know what locking is used at the lower level; but if it’s a filesystem that supports quotas, it will be using i_lock as in inode_add_bytes().“*[7]

Abschließend ist festzuhalten: Die Sperren-Dokumentation des Linux-Kerns ist in einem fragwürdigen Zustand. Weder wird eine bestimmte Syntax zum Beschreiben der Regeln noch eine präzise Beschreibung der Sperren verwendet. Außerdem zeigen die Kommentare, dass es einen hinreichenden Bedarf gibt, Klarheit zu schaffen, welche Sperren an welcher Stelle zu nutzen sind.

6 Entnommen aus `linux-v5.6/fs/inode.c`, Zeile 812.

7 Ein Auszug eines Kommentars aus der Funktion `fsstack_copy_inode_size()` in `linux-v5.6/fs/stack.c`.

```
/*
 * inode_set_flags - atomically set some inode flags
 *
 * Note: the caller should be holding i_mutex, or else be sure that
 * they have exclusive access to the inode structure (i.e., while the
 * inode is being instantiated).  The reason for the cmpxchg() loop
 * --- which wouldn't be necessary if all code paths which modify
 * i_flags actually followed this rule, is that there is at least one
 * code path which doesn't today so we use cmpxchg() out of an abundance
 * of caution.
 *
 * In the long run, i_mutex is overkill, and we should probably look
 * at using the i_lock spinlock to protect i_flags, and then make sure
 * it is so documented in include/linux/fs.h and that all code follows
 * the locking convention!!
 */
```

Listing 3.2: Dokumentation der Funktion `inode_set_flags` aus der Datei `linux-v5.6/fs/inode.c`. (Nach einer Vorlage von Lochmann et al. [LSBS19])

3.2 FREEBSD

Der Betriebssystemkern von FreeBSD [MNNW14] stellt ebenfalls verschiedene Mechanismen zur Synchronisation verschiedener Arten von Nebenläufigkeit bereit [MNNW14]. Darüber hinaus durchlief er eine vergleichbare Entwicklung bzgl. der Multiprozessorsynchronisation wie der Linux-Kern: Auch hier gab es zunächst eine Sperre, die den Kern gegen konkurrierende Zugriffe von mehreren Prozessorkernen absicherte [Leho1]. Dies wurde aber zu Gunsten einer höheren Parallelität durch ein feingranulareres Sperren ersetzt [Leho1]. Ähnlich zu Abbildung 1.1 stellt Abbildung 3.1 die Entwicklung des FreeBSD-Kerns dar: Wie zu sehen ist, stieg die Codegröße in den letzten 10 Jahren um 75,09 %. Es wird die Verwendung von drei verschiedenen Sperren-Typen dargestellt. Analog zu Linux wurde die Häufigkeit der Aufrufe der Initialisierungsfunktionen der betreffenden Sperren-Typen gezählt. Auch hier liegt die Annahme zugrunde, dass die Beliebtheit eines Sperren-Typs mit der Häufigkeit der Initialisierungsaufrufe korreliert.

Im FreeBSD-Kern ist die Nutzung von Sperren des Typs *sxlock* um 132,97 % gestiegen. Analog verhält es sich bei den zwei Varianten einer Mutex *mutexsleep* und *mutexspin*. Auch deren Nutzung stieg um 72,73 % respektive 49,58 %. Die mitwachsende Nutzung von Sperren zeigt, dass das feingranulare Sperren auch im FreeBSD-Kern an Bedeutung behält. Dies wird nochmal durch die um 93,73 % gestiegene Anzahl an Kommentaren[8] , die in Verbindung Synchronisation stehen, betont. Neben den bereits erwähnten drei Typen gibt es noch vier weitere Typen, so dass einem Entwickler in der Größenordnung sieben verschiedene Sperren-Typen[9] zur Verfügung stehen [MNNW14].

8 Regulärer Ausdruck zur Suche nach synchronisations-relevevanten Quellcodekommentaren: `<racing>|<race>|protected by|caller holds|protects|should be holding|called under|must be held|lock ordering|lock <order>`.

9 Die genaue Anzahl hängt letztlich von der Kategorisierung ab.

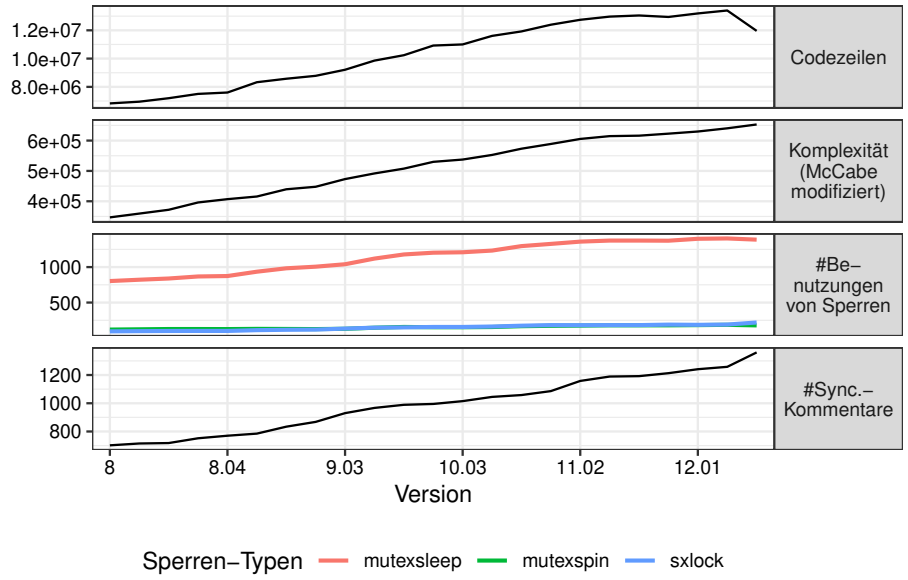


Abbildung 3.1: Die Entwicklung der Codegröße sowie der Code-Komplexität des FreeBSD-Kerns über die letzten 10 Jahre sowie die Nutzungshäufigkeit von drei verschiedenen Sperren-Typen über denselben Zeitraum. Ebenso werden die Anzahl der synchronisations-relevanten Quellcodekommentare dargestellt.

```
1  /*
2   * [...]
3   * Lock reference:
4   *  l - mnt_listmtx
5   *  m - mountlist_mtx
6   *  i - interlock
7   *  v - vnode freelist mutex
8   *
9   * Unmarked fields are considered stable as long as a ref is held.
10  *
11  */
```

Listing 3.3: Ein Auszug aus der Sperren-Dokumenation der Datenstruktur `struct mount` aus dem FreeBSD-Kern – entnommen aus `freebsd-13.0/sys/sys/mount.h`.

Für diese Sperren-Typen existiert ebenfalls Literatur, die sich allgemein der Einsatzmöglichkeiten und der Eigenschaften widmet [MNNW14]. Es fehlt jedoch die Zuordnung zwischen Datenstrukturen und Sperren im Kern. Diese Lücke vermag der Quellcode in Form einer deutlich detaillierteren und zentralen Dokumentation der unterschiedlichen Datenstrukturen zu schließen. Nach dem derzeitigen Kenntnisstand des Autors enthalten einige, aber nicht alle Definitionen von zentralen Datenstrukturen des FreeBSD-Kerns Informationen, welche Sperren für den Zugriff auf ein Element der jeweiligen Datenstruktur erforderlich sind. Hierzu befindet sich vor der Definition der Datenstruktur ein größerer Kommentarblock, der die an der Datenstruktur beteiligten Sperren auflistet und ihnen Buchstaben zuordnet. Ein Beispiel

```
1  /*
2   * Reading or writing any of these items requires holding the appropriate
        lock.
3   *
4   * Lock reference:
5   *  c - namecache mutex
6   *  i - interlock
7   *  l - mp mnt_listmtx or freelist mutex
8   *  I - updated with atomics, 0->1 and 1->0 transitions with interlock held
9   *  m - mount point interlock
10  *  p - pollinfo lock
11  *  u - Only a reference to the vnode is needed to read.
12  *  v - vnode lock
13  *
14  * Vnodes may be found on many lists.  The general way to deal with operating
15  * on a vnode that is on a list is:
16  *  1) Lock the list and find the vnode.
17  *  2) Lock interlock so that the vnode does not go away.
18  *  3) Unlock the list to avoid lock order reversals.
19  *  4) vget with LK_INTERLOCK and check for ENOENT, or
20  *  5) Check for DOOMED if the vnode lock is not required.
21  *  6) Perform your operation, then vput().
22  */
```

Listing 3.4: Ein Auszug aus der Sperren-Dokumenation der Datenstruktur `struct vnode` aus dem FreeBSD-Kern – entnommen aus `freebsd-13.0/sys/sys/vnode.h`.

ist in Listing 3.4 für die Datenstruktur `struct vnode` zu sehen. Ab Zeile 3.3 werden acht verschiedene Synchronisationsmechanismen für diese Datenstruktur definiert und mit Buchstaben versehen. Darüber hinaus wird in Zeile 14 ff. noch auf Besonderheiten beim Sperren hingewiesen. Wenngleich sie an einer zentralen Stelle dokumentiert sind, so erfolgt ihre Beschreibung in einem Fließtext, der grundsätzlich Spielraum für Interpretationen lässt. Eine Übertragung in ein maschinenlesbares Format wird erschwert. Die grobe Struktur der verschiedenen Kommentarblöcke sind zwar ähnlich, die konkrete Struktur bei einer bestimmten Datenstrukturen weist jedoch Unterschiede auf: In Zeile 11 in Listing 3.4 wird die Tatsache, dass anstatt einer Sperre nur eine Referenz auf die Instanz erforderlich ist, als eigene Sperre in dem Kommentarblock ausgedrückt. Bei der Datenstruktur `struct mount` wird dieselbe Tatsache lediglich als Kommentar am Ende der Dokumentation erwähnt – vgl. Zeile 9 in Listing 3.3. Listing 3.5 zeigt ein Beispiel für die Verwendung der zuvor definierten Zuordnung. In Zeile 6 wird für das Element `v_mount` beispielsweise festgelegt, dass keinerlei Sperren beim Zugriff erforderlich sind. Zeile 7 hingegen beschreibt, dass auf das Element `v_nmntvnodes` nur zugegriffen werden darf, während eine dedizierte Sperre gehalten wird. Somit ist eine für den Entwickler eindeutige Zuordnung der jeweiligen Sperren zu den Elementen der Datenstruktur geschaffen worden. Dies bietet grundsätzlich ein großes Potential: Es würde die automatische Extraktion und anschließende Verifikation der Sperren-Regeln erlauben. Allerdings zeigt sich bei der Umsetzung dieses Schemas kein einheitliches Bild.

```
1 struct vnode {
2     // [...]
3     /*
4      * Filesystem instance stuff
5      */
6     struct  mount *v_mount;         /* u ptr to vfs we are in */
7     TAILQ_ENTRY(vnode) v_nmntvnodes;    /* m vnodes for mount point */
8     // [...]
9 };
```

Listing 3.5: Ein Auszug aus der der Datenstruktur `struct vnode` aus dem FreeBSD-Kern – zu finden in `freebsd-13.0/sys/sys/vnode.h`. Hinter jedem Element einer Datenstruktur wird mit Hilfe des zu vor festgelegten Alphabets dokumentiert, welche Sperre beim Zugriff zu verwenden ist.

Teilweise wird der Variablenname der Sperre verwendet (vgl. Zeile 7 in Listing 3.4), manchmal jedoch nur eine Beschreibung, wie in Zeile 12. Letzteres schmälert den Vorteil der hier eingesetzten systematischen Dokumentation.

Dennoch zeigt FreeBSD im Vergleich zu Linux ein deutlich besseres Bild, was sowohl den Zustand als auch die Form der existierenden Sperren-Dokumentation betrifft. Dennoch, wie oben erwähnt, gibt es bei der Syntax und der Nennung der Sperren Ungenauigkeiten.

3.3 ZUSAMMENFASSUNG

In den beiden vorangegangenen Abschnitten wurde der Stand der Sperren-Dokumentation der Betriebssystemkerne von Linux wie FreeBSD dargestellt. Der aktuelle Zustand der Dokumentation, wenngleich nicht alle Beobachtungen bei den Systemen gleichermaßen zutreffen, lässt sich wie folgt zusammenfassen: Die Sperren-Dokumentation

- ist über den Quellcode verstreut,
- unvollständig oder alt,
- die Regeln folgen keiner einheitlichen Syntax und
- die Beschreibungen der zu verwendeten Sperren sind nicht präzise.

Es ist für den Entwickler somit schwierig, alle Informationen über die für seine Aufgabe nötigen Sperren zu ermitteln [LSBS19]. Dies gilt insbesondere, wenn er nicht der Originalautor des Quellcodes ist [LSBS19]. Unter dem Strich erhöht dies die Wahrscheinlichkeit von Programmierfehlern [LSBS19].

Aus der beschriebenen Situation ergibt sich unweigerlich die Frage, ob die existierende Dokumentation noch zu dem Quellcode passt [LSBS19]. Woraus sich die Notwenigkeit ableiten lässt, die existierende Sperren-Dokumentation auf ihre Korrektheit zu überprüfen [LSBS19].

Zusätzlich ist es durch die steigende Nutzung von Sperren in beiden Betriebssystemkernen sowie durch die grundsätzliche Vergrößerung des Programmcodes wünschenswert, eine Sperren-Dokumentation sowohl für exis-

tierende als auch für neue Datenstrukturen zu haben [LSBS19]. Somit muss es die Möglichkeit geben, neue Dokumentation zu erzeugen. Aufgrund der Größe der Projekte sollte dies natürlich ein automatisierter Vorgang sein.

Zur Steigerung des Mehrwerts der generierten Dokumentation bietet es sich an, den Quellcode abschließend gegen die neue Dokumentation zu testen. Nur so ist sichergestellt, dass die Dokumentation und der Quellcode zu einander passen. In der Konsequenz trägt dies zur einer Verbesserung des Betriebssystems bei.

Abschließend sind folgende drei Anforderungen an eine automatisierte Lösung zur Sperren-Analyse in Betriebssystemen zu stellen [LSBS19]:

- Das Überprüfen von existierender Sperren-Dokumentation hinsichtlich ihrer Korrektheit.
- Das automatische Generieren von neuer Sperren-Dokumentation, die einer eindeutigen Syntax folgt.
- Das Überprüfen des existierenden Quellcodes hinsichtlich der Einhaltung der neuen Sperren-Dokumentation.

4

VERWANDTE ARBEITEN

Inhalt

Kapitel 3 zeigte auf, welche drei Anforderungen an eine automatische Sperren-Analyse zu stellen sind [LSBS19]:

- Das Überprüfen von existierender Sperren-Dokumentation hinsichtlich ihrer Korrektheit.
- Das automatische Generieren von neuer Sperren-Dokumentation, die einer definierten Syntax folgt.
- Das Überprüfen des existierenden Quellcodes hinsichtlich der Einhaltung der neuen Sperren-Dokumentation.

Der Ausgangspunkt für alle drei Anforderungen ist die Bestimmung der gehaltenen Sperren, die sog. Sperren-Menge[1], zum Zeitpunkt der Speicherzugriffe auf einen beobachteten Speicherbereich, also ein Element einer komplexen Datenstruktur oder eine Instanz eines einfachen Datentyps. Die Zugriffe auf diese Datenstruktur selbst müssen natürlich auch festgehalten werden. Aus dieser Information lassen sich weitere Schlüsse wie z. B. die Menge der gehaltenen Sperren über alle Zugriffe oder aber sich widersprechende Sperren-Mengen ableiten. Daher wird im weiteren Verlauf dieses Kapitels ein Blick auf die verwandten Arbeiten unter dem Gesichtspunkt der Bestimmung der Sperren-Mengen geworfen.

Eine der drei eingangs genannten Anforderungen betrifft die Generierung neuer Sperren-Dokumentation. Die automatische Erzeugung von Quellcode-Dokumentation im Allgemeinen ist Bestandteil existierender Arbeiten: McBurney et al. beispielsweise analysieren Funktionen sowie deren Aufrufkontext mit Hilfe von statischer Analyse in *JAVA* [MM14]. Sie erzeugen daraus Quellcode-Dokumenation in natürlicher Sprache [MM14]. Hu et al. nutzen hingegen Techniken des maschinellen Lernens, um mittels Quellcode-Annotationen und dem Programmcode selbst Dokumentation zu generieren [HLX+18]. Da sich nach derzeitigem Wissensstand des Autors jedoch keine Arbeiten finden lassen, die sich entweder mit der Generierung oder mit der

1 Deutsch für *lock set* [SBN+97]

Überprüfung der Sperren-Dokumentation von (System-)Software auseinandersetzen, widmet sich dieses Kapitel lediglich dem vorgenannten Punkt über die Sperren-Mengen.

Orthogonal zu dem beschriebenen Themenkomplex sind die Ansätze zur formalen Verifikation von kompletten Betriebssystemen [Bev89, KEH+09, Kle09] oder Teilen davon [Wit07, CZC+15, RST+15, dOCdO19] zu sehen. Mit ihrer Hilfe lassen sich etwaige Programmierfehler von vorne herein ausschließen: Es wird formal gezeigt, dass der Programmcode (oder Teile davon) korrekt arbeitet. Die Machbarkeit einer formalen Verifikation hängt allerdings von verschiedenen Faktoren wie z. B. der Komplexität des Systems ab: „The main factors that influence the practicability of formal correctness proofs are the level of detail, the complexity, and the size of a program." [Kle09]. Da diese Arbeit jedoch die aufzeichnungsbasierte Analyse von Systemsoftware behandelt, werden die Arbeiten aus diesem Feld hier nicht weiter vorgestellt.

Ebenfalls komplementär zu dem hier präsentierten Ansatz ist der Einsatz von sicheren Programmiersprachen in der Betriebssystementwicklung [BSP+95, vECC+99, BJV03]. In der jüngeren Vergangenheit wurde die Programmiersprache *Rust* [KN19] zur Betriebssystementwicklung untersucht [LAC+15, LBP19, NHD+20]. *Rust* besitzt Sprachmittel um Wettlaufsituationen im Speicher zu verhindern, so dass mögliche Verstöße zur Übersetzungszeit aufgedeckt werden [KN19]. Ebenso ist das Absichern und der Umgang mit geteilten Daten in konkurrierenden Kontrollflüssen Bestandteil der Sprache [KN19]. Auch hier findet zur Übersetzungszeit eine Überprüfung statt [KN19]. Ferner ist ohne eine explizite Deklaration der direkte Zugriff auf Adressen sowie der Hardwarezugriff verboten [KN19].

Der Abschnitt 4.1 befasst sich mit Arbeiten, die zur Bestimmung der Sperren-Mengen den Quellcode analysieren. Wohingegen der Abschnitt 4.2 Arbeiten sowie Werkzeuge zur Laufzeitanalyse von Software vorstellt. Den Abschluss bildet in Abschnitt 4.3 die Kategorie der Arbeiten, die eine nachträgliche Analyse durchführen.

4.1 STATISCHE ANALYSE

Die statische Analyse von Quellcode dient dem Auffinden und ggf. Extrahieren von Mustern in Programmcode. Dabei kann entweder intraprozedural oder interprozedural vorgegangen werden [Hin01]. Bei ersterem wird lediglich der Quellcode innerhalb einer Funktion auf das gewünschte Muster untersucht. Wird eine weitere Funktion aufgerufen, so wird deren Inhalt nicht weiter beachtet. Muster, die sich über mehrere Funktionen erstrecken, sind so offensichtlich nicht zu finden. An dieser Stelle kann ein interprozedurales Vorgehen Abhilfe schaffen [Hin01]. Während der Analyse kann mit Hilfe des statisch-ermittelten Kontrollflussgraphen nach Mustern funktionsübergreifend gesucht werden, was jedoch die Komplexität erhöht [Hin01]. Darüber hinaus trägt der Einsatz von Zeigern, wie z. B. in der Programmiersprache *C*, zusätzlich zur Komplexität bei [Hin01]: Da der Wert eines Zeigers im Vorfeld

nicht immer bestimmt werden kann, muss die Analyse alle möglichen Werte durchgehen. Dieses Problem ist jedoch unentscheidbar [Lan92, Ram94]. Daher versuchen andere Arbeiten die Abschätzung des Werteraums zu vereinfachen [Hin01, LLA07, HL07]: Engler und Ashcraft berücksichtigen beispielsweise beim Auffinden von Zielen von Funktionszeigern die Signatur der Funktion [EA03].

Engler et al. waren es auch, die sich in ihrer Arbeit einen allgemeinen Ansatz basierend auf statischer Code-Analyse zum Auffinden von Programmfehlern[2] erarbeiteten [ECH+01]. Sie leiten anhand des Quellcodes die Intention des Programmierers, „programmers belief“ [ECH+01], ab und versuchen anhand dessen Widersprüche im Programmcode zu finden [ECH+01]. Sie gehen grundsätzlich davon aus, dass Programmfehler ein abweichendes Verhalten, „Bugs as deviant behavior [...]“ [ECH+01], darstellen [ECH+01]. Somit wird angenommen, dass das normale Verhalten die Regel darstellt. Eine exemplarische Annahme ist etwa, dass ein bestimmter Zeiger eine valide Adresse enthält, wenn er an einer bestimmten Stelle im Programmcode dereferenziert wird [ECH+01]. Engler et al. durchsuchen den betreffenden Quellcode nun nach Stellen, die dieser Annahme widersprechen [ECH+01]. Engler et al. wenden ihren Ansatz auf den Linux- und OpenBSD-Kern an [ECH+01]. Außerdem geben sie ein Beispiel, wie man mit dieser Methode herausfinden kann, ob an eine Sperre überall korrekt eingesetzt wird [ECH+01]. Zum Einsatz kommt dieses Beispiel jedoch nicht [ECH+01].

Darauf aufbauend entwickeln Engler und Ashcraft einen Ansatz zur Detektion von Verklemmungen und von Wettlaufsituationen zwischen Speicherzugriffen [EA03]. Hierzu untersuchen sie den Kontrollflussgraphen des betreffenden Systems, hier u.a. der Linux- und FreeBSD-Kern, auf verdächtige Stellen [EA03]. Ihre Ergebnisse sortieren sie so, dass die vielversprechendsten Kandidaten oben stehen, dabei wird je eine Liste für Wettlaufsituationen und für Verklemmungen erstellt [EA03]. Wenngleich es ihnen gelingt mit ihrem Ansatz sechs bestätigte Verklemmungen in beiden Betriebssystemkernen und drei bestätigte Wettlaufsituationen in Linux zu finden [EA03], so gibt es doch Einschränkungen: Sie führen keine Alias-Analyse durch [EA03]. Bei Zeigern auf Datenstrukturen wird vereinfachend angenommen, dass es sich dabei um eine Variable (anstatt eines Zeigers) von dem entsprechenden Typ handelt [EA03]. Lediglich für Funktionszeiger wird eine simple Analyse durchgeführt: Engler und Ashcraft betrachten beim Aufruf eines Funktionszeigers alle Funktionen als mögliche Ziele, die zu dem Typ passen und jemals irgendwo einem Funktionszeiger zugewiesen wurden [EA03].

Die beiden bisher geschilderten sowie andere Arbeiten [VJL07, HK13] erwecken in der Breite einen guten Eindruck, da sie viele Anwendungsfälle abdecken. Jedoch weisen sie im Detail, gerade bei der Alias-Analyse, Schwächen auf. Daher werden im Folgenden Arbeiten vorgestellt, die weniger in der Breite agieren, dafür eine bessere Präzision aufweisen. Dies wird durch eine Maßschneiderung auf beispielsweise ein bestimmtes Subsystem

2 Deutsch für *Bugs*

erreicht. Exemplarisch werden zwei Arbeiten vorgestellt, die sich auf die Gerätetreiber von Linux beschränken. Der darauf folgende Absatz gibt noch einen Überblick über andere Werkzeuge zur Sperren-Analyse.

ANALYSE VON GERÄTETREIBERN Lu et al. führen in ihrer Arbeit eine statische Analyse von vier großen Softwareprojekten hinsichtlich der Wettlaufsituationen zwischen Speicherzugriffen durch [LPH+07]. Dazu untersuchen sie den Quellcode über Funktionsgrenzen hinweg auf Korrelationen zwischen Variablenzugriffen, wie z. B. „Immer wenn Variable *X* gelesen wird, wird im Anschluss Variable *Y* geschrieben." (übersetzt) [LPH+07]. Darauf aufbauend detektieren Lu et al. zwei Arten von Fehlern: a) inkonsistente Aktualisierungen von mehreren zusammenhängenden Variablen und b) fehlerhafter Umgang mit Nebenläufigkeit von zusammenhängenden Variablen [LPH+07]. Der erste Fall umfasst zwei oder mehr Varibalen, die – gemäß Analyse – immer zusammen zugegriffen werden [LPH+07]. Geschieht dies an einer Stelle nicht, wird dies als potentieller Fehler angesehen [LPH+07]. Als Beispiel sei das Vergrößern eines Feldes ohne das Erhöhen der zugehörigen Längenvariable genannt [LPH+07]. Im zweiten Fall werden Speicherzugriffe detektiert, bei denen die Schnittmenge der jeweils gehaltenen Sperren leer ist, die vorhergehende Analyse aber gezeigt hat, dass sie zusammengehören [LPH+07]. Neben *Mozilla Firefox*, *MySQL* und *PostgreSQL* wendeten sie ihren Ansatz auf die Gerätetreiber des Linux-Kerns an [LPH+07]. Die Ausgangsanalyse sowie das Finden von Korrelationen zwischen von Variablenzugriffen untersuchen Lu et al. in allen vier Projekten [LPH+07]. Lediglich der Linux-Kern wird auf die zweite Fehlerart nicht untersucht [LPH+07]. In ihrer Alias-Analyse nehmen die Autoren an, dass Zeiger-Aliasse ihre Analyse nicht beeinträchtigen, da sie Korrelationen zwischen Zugriffen auf Elemente von Datenstrukturen berücksichtigen [LPH+07]. Dies lässt sich anhand eines Beispiels aus dem Linux-Kern widerlegen: Wie in Abbildung 4.1 zu sehen ist, enthält die Datenstruktur `struct inode` u.a. ein Element des Typs `struct list_head`. Dies wird innerhalb des Linux-Kerns eingesetzt, um Listen mit beliebigen Inhalten zu erzeugen, wenngleich eine konkrete Liste natürlich nur Elemente eines Datentyps enthält. Das Element `i_lru` enthält hierbei die für die Verkettung notwendigen Zeiger. Sie zeigen jedoch nicht auf das Objekt selbst, sondern, wie in Abbildung 4.1 zu sehen, auf das Element `i_lru`. Bei allen Listenoperationen sind die verwendeten Zeiger folglich vom Typ `struct list_head`. Wenngleich die Zeiger in `struct list_head` auf Objekte vom Typ `struct inode` zeigen, ist diese Beziehung anhand des Quellcodes nicht ablesbar. Daher würde die Analyse von Lu et al. hier keinerlei Aliasing feststellen. Daher legt die vorgenannte Aussage von Lu et al. nahe, dass sie keine Alias-Analyse für Zeiger durchführen.

Hier setzt die Arbeit von Vojdani et al. an: Sie untersuchen mittels statischer Analyse die Gerätetreiber im Linux-Kern auf Wettlaufsituationen zwischen Speicherzugriffen [VAR+16]. Sie behandeln dabei Werte-abhängige Synchronisation [VAR+16]. Ebenso unterscheiden sie zwischen a) Sperren, die Instanzen von Datentypen oder Teile davon schützen, b) gröberen Syn-

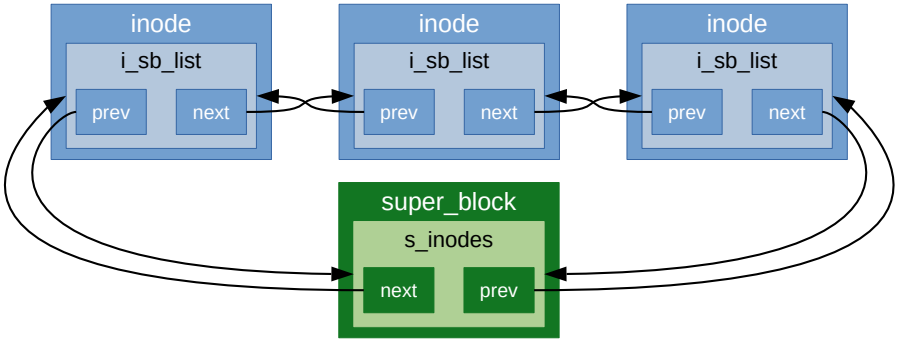


Abbildung 4.1: Graphische Darstellung der Beziehung zwischen den Datentypen `struct inode` (Definiert in `linux-v5.6/include/linux/fs.h`, Zeile 628 ff.) und `struct list_head`. Die Datenstruktur `struct list_head` ist als Element in die Datenstruktur `struct inode` eingebettet. Sie dient dem Einhängen von `struct inode` in eine Liste – hier ist es die Liste aller Inodes zu einem Superblock. Die Verkettung erfolgt dabei durch Zeiger auf das Element `i_sb_list`, nicht auf das gesamte Objekt. Der Kopf der Liste wird in `s_inodes` in der Datenstruktur `struct super_block` (Definiert in `linux-v5.6/include/linux/fs.h`, Zeile 1425 ff.) gespeichert. (Illustration nach einer Darstellung von Voldani et al. [VAR+16])

chronisationsmustern, wie z. B. eine Sperre für jede Liste, c) Feldern von Sperren oder d) dem bereits in Abbildung 4.1 dargestellten Szenario [VAR+16]. Sie betrachten bei ihrer Analyse jedoch keine Funktionsaufrufe [VAR+16].

In einer anderen Arbeit untersuchen Bai et al. die Linux-Gerätetreiber auf nebenläufige *use-after-free*-Fehler [BLCH19]. Hierbei gibt ein Kontrollfluss einen Speicherinhalt frei, wohingegen ein anderer Kontrollfluss nebenläufig auf diesen zugreift – unabhängig davon, ob der Kontrollfluss eine Sperre einsetzt [BLCH19]. Für ihre Arbeiten nutzen Bai et al. die Struktur von Gerätetreibern unter Linux aus: Ein Gerätetreiber besteht aus einer oder mehreren Quellcodedateien sowie – je nach Gerätetyp – einer Menge von Funktionen, die in Form von Funktionszeigern an den Kern als Schnittstelle weitergereicht werden [BLCH19]. Zunächst finden sie innerhalb eines Treibers Paare aus potentiell parallel ausführbaren Schnittstellen-Funktionen [BLCH19]. Hierbei finden sie pro Treiber alle möglichen Paare an Schnittstellenfunktionen, die potentiell parallel ausgeführt werden können [BLCH19]. Funktionszeiger werden hierbei nicht berücksichtigt [BLCH19]. Anhand der Häufigkeit eines einzelnen Funktionspaares über alle Treiber wird bewertet, wie relevant das jeweilige Paar ist [BLCH19]. Für alle relevanten Paare, bei denen nun auf die gleichen Variablen zugegriffen wird, die Sperren-Menge leer ist und eine Funktion zur Speicherfreigabe aufgerufen wird, werden nun Warnungen erzeugt [BLCH19]. Für die Bestimmung der Sperren machen sie sich zunutze, dass die relevanten Sperren meist in einer für den Treiber zentralen Datenstruktur eingebettet sind, so dass keine Alias-Analyse nötig ist [BLCH19].

```
1  static int expand_fdtable(struct files_struct *files, unsigned int nr)__
2      releases(files->file_lock)__
3      acquires(files->file_lock)
4  {
5      // [...]
6      spin_unlock(&files->file_lock);
7      // [...]
8      spin_lock(&files->file_lock);
9      // [...]
10     return 1;
11 }
```

Listing 4.1: Beispiel für die Annotation von Funktionen im Linux-Kern zur späteren Analyse mittels *Sparse* [Lin18, Jon04, Lin04, Nei16]. Hier dargestellt sind die wesentlichen Aspekte der Funktion `expand_fdtable` aus der Datei `linux-v5.6/fs/file.c`.

VERSCHIEDENE WERKZEUGE Breuer et al. präsentieren in ihrer Arbeit eine Methode zur statischen Analyse von *C*-Programmcode im Linux-Kern hinsichtlich der falschen Benutzung von Sperren des Typs *spinlock* [BV04]. Ihr Ansatz findet Stellen im Programmcode, die zu einer Verklemmung führen [BV04]. Darunter ist die Situation zu verstehen, dass eine potentiell schlafende Funktion aufgerufen wird, während eine Sperre gehalten wird [BV04]. Abschließend diskutieren sie noch weitere Einsatzmöglichkeiten ihres Ansatzes: Dazu zählt das Berücksichtigen von mehreren Sperren sowie deren Reihenfolge [BV04]. Außerdem wollen sie die Informationen über Sperren verfeinern, indem sie vermerken, ob eine Sperre ein globales Objekt oder aber in eine andere Datenstruktur eingebettet ist [BV04]. Eine direkte Beziehung zwischen Datentypen und Sperren stellen sie jedoch nicht her [BV04].

Das Werkzeug *SLAM* führt eine statische Analyse von *C*-Code durch, um falsche Benutzungen von Programmierschnittstellen[3] aufzudecken [BR02]. Hierbei ist keinerlei Annotation des bestehenden Quellcodes seitens des Programmierers erforderlich [BR02]. Es muss lediglich die zu überprüfende Sicherheitseigenschaften, wie z. B. „[...] a lock should be alternatingly acquired and released“ [BR02], spezifiziert werden [BR02]. Ihren Ansatz haben Ball et al. auf die Gerätetreiber von Windows XP angewendet [BR02]. Abgesehen von der Möglichkeit, ein Ungleichgewicht bei der Verwendung der Programmierschnittstelle von Sperren zu detektieren, widmen sich die Autoren nicht dem weiteren Umgang mit Sperren [BR02].

Ein weiteres Werkzeug zur statischen Code-Analyse ist *Sparse* [Lin18, Jon04, Lin04, Nei16]. Es ging aus der Entwicklung des Linux-Kerns hervor [Lin18, Jon04]. Dort wurde es zunächst zum Auffinden von unsicherem Umgang mit Zeigern auf Speicherbereiche in der Anwendungsebene genutzt [Lin18, Jon04]. Mittlerweile hat sich daraus ein breiterer Anwendungsbereich entwickelt: Es wird beispielsweise eingesetzt, um auf Funktionsebene ein Ungleichgewicht bei der Verwendung der Programmierschnittstelle von Sperren zu finden – wie in der entsprechenden Dokumentation im Linux-Kern

3 Deutsch für *application programming interfaces*, kurz *API*

dargestellt ist [Lino4, Nei16, Lin19]. Es geht dabei davon aus, dass für jede verwendete Sperre a) die Anzahl an *p()* und *v()* Operationen zwischen Funktionseintritt und bei Funktionsaustritt gleich ist und b) die logisch korrekte Reihenfolge dieser Operationen eingehalten wird [Lino4, Nei16, Lin19]. Ist dies nicht der Fall, so zeigt es dies durch einen Fehler an [Lino4, Nei16, Lin19]. Ist es jedoch vom Programmierer explizit gewünscht, von diesen Vorgaben abzuweichen, so kann er dies durch Annotationen an dem Funktionskopf ausdrücken [Lino4, Nei16, Lin19]. In Listing 4.1 ist ein Beispiel für eine solche Annotation anhand der Funktion `expand_fdtable()` gegeben. Entgegen der Grundannahme von *Sparse* wird in Zeile 6 die Sperre erst freigegeben und anschließend in Zeile 8 wieder angefordert. Da diese Abweichung vom Entwickler beabsichtigt ist, wird in Zeile 2 bzw. 3 eine entsprechende Annotation vorgenommen. Hierdurch wird *Sparse* keinerlei Fehlermeldung ausgeben. Anhand der Dokumentation ist jedoch nicht eindeutig klar, ob die Analyse funktionsübergreifend durchgeführt wird – sie lässt es jedoch vermuten [Lino4, Nei16, Lin19]. Mit diesem Werkzeug lassen sich falsche Verwendungen von Sperren aufdecken. Allerdings macht die Dokumentation keinerlei Aussagen darüber, ob die Sperren zum richtigen Zeitpunkt bezüglich des Speicherzugriffs geholt werden. Eine Zuordnung zwischen Sperren und Datenstrukturen ist somit nicht möglich. Ferner kann *Sparse* nicht zwischen verschiedenen Sperren unterscheiden [Nei16].

4.2 ANALYSE ZUR LAUFZEIT

Im vorangegangenen Abschnitt 4.1 wurde das Feld der statischen Code-Analyse beleuchtet. Hierbei findet die Untersuchung ohne eine Ausführung des betreffenden Programmcodes statt. In diesem Abschnitt soll der Blick auf die Arbeiten gerichtet werden, die sich mit der Analyse in Ausführung befindlicher Software befassen.

Eine frühe und bedeutende Arbeit in diesem Bereich stammt von Stefan Savage et al. [SBN+97]. In ihrem *Eraser* genannten Ansatz untersuchten sie allgemein parallele Software [SBN+97]. Dabei nutzen Savage et al. den von ihnen erarbeiteten *LockSet*-Algorithmus zur Detektion von Wettlaufsituationen zwischen Speicherzugriffen [SBN+97]. Hierzu wird bei jedem Speicherzugriff die Schnittmenge aus der Menge der bisher beobachteten Sperren für eine bestimmte Adresse mit der Menge der durch den aktuellen Kontrollfluss gehaltenen Sperren gebildet [SBN+97]. Erhalten sie hierbei als Ergebnis die leere Menge, so wird eine Warnung ausgegeben, die auf eine mögliche Wettlaufsituation hinweist [SBN+97]. Ihr Algorithmus berücksichtigt ebenfalls die Initialisierungsphase von Datenstrukturen, wo zumeist keinerlei Sperren erforderlich sind, sowie nur lesend geteilte Daten und Leser-Schreiber-Sperren [SBN+97]. Allerdings können Wettlaufsituationen nicht nur zustande kommen, wenn keinerlei Sperren gehalten werden, sondern auch wenn zu wenige Sperren gehalten werden. Dieser Fall wird von *Eraser* jedoch nicht abdeckt, da der Algorithmus davon ausgeht, dass der Zugriff durch zumindest eine Sperre geschützt ist [SBN+97].

Ebenfalls auf Anwendungsebene operiert das Werkzeug *Valgrind* [NS07, SNW08]. *Valgrind* selbst ist ein Rahmenwerk zur dynamischen Instrumentierung von Binärcode [NS07, SNW08]. Die eigentliche Funktionalität wird durch die jeweilige Erweiterung umgesetzt. In dem konkreten Fall implementiert die Erweiterung *DRD* [4] einen Mechanismus zur Detektion u.a. von Wettlaufsituationen zwischen Speicherzugriffen in mehrfädigen Programmen, die falsche Verwendung der Programmierschnittstelle von *POSIX*-Synchronisationsmitteln sowie von Flaschenhälsen beim Holen von Sperren. Das Auffinden von Wettlaufsituationen erkennt beispielsweise die Semantik von Leser-Schreiber-Sperren: Wird beim Schreiben einer Speicherstelle nur eine Leser-Sperre gehalten, so wird ein Fehler ausgegeben[5]. Ein ähnliches Werkzeug ist mittlerweile Bestandteil von *clang*[6], dem *C/C++*-Compiler von LLVM [SI09, SPIV12].

Ebenso befassen sich Agarwal et al. sowie Jula et al. mit dem Detektieren von Verklemmungen bzw. Härten gegen Verklemmungen in mehrfädigen Anwendungen [AS06, JTZC08].

Neben den bisher vorgestellten Arbeiten, die vornehmlich Anwendungsprogramme untersuchen, gibt es ebenso Arbeiten, die sich mit dem Betriebssystemkern befassen: Erickson et al. untersuchen beispielsweise mit Hilfe ihres Ansatzes *DataCollider* den Betriebssystermkern von Windows 7 auf Wettlaufsituationen zwischen Speicherzugriffen [EMBO10]. Sie nutzen hierzu spezielle Prozessorregister, um die Speicherzugriffe zu beobachten [EMBO10]. Ein ähnliches Vorgehen wählen Jiang et al. [JYX+16]. Sie beschränken sich bei ihrer Untersuchung jedoch auf zwei Dateisysteme im Linux-Kern [JYX+16].

Chen et al. detektieren ebenfalls Wettlaufsituationen bei Speicherzugriffen [CBJ+19]. Sie berücksichtigen hingegen mit Hilfe des *LockSet*-Algorithmus' die gehaltenen Sperren [CBJ+19]. Dazu instrumentieren sie zum Übersetzungszeitpunkt u.a. sowohl die Speicherzugriffe als auch Operationen auf den Sperren [CBJ+19]. Dies nutzen sie, um zur Laufzeit die entsprechenden Informationen im Gerätetreiber zu erheben [CBJ+19]. Diese werden abschließend genutzt, um entsprechende Fehler im Programmcode zu finden [CBJ+19].

Abseits von den vorgenannten Arbeiten, die sich zumeist mit Wettlaufsituationen bei Speicherzugriffen befassen, widmen sich folgende Arbeiten der Performanz beim Sperren: Exemplarisch sind hier *HaLock* [HCC+12], *LockMeter* [BH00] sowie die Arbeit von Chen et al. [CS12] zu nennen. Sie befassen sich alle mit dem Auffinden von Flaschenhälsen in kritischen Abschnitten, wenn zu viele Kontrollflüsse die Sperre anfordern. Für den Ansatz *HaLock* ist darüber hinaus noch Hardwareunterstützung erforderlich [HCC+12]. Lediglich die Arbeit *LockMeter* befasst sich dabei dem Betriebssystemkern von Linux [BH00].

---

4 https://sourceware.org/git/?p=valgrind.git;a=blob;f=drd/docs/drd-manual.xml;hb=555ddc4753e9014b809683c840867ed424dd7d99

5 Vgl. https://sourceware.org/git/?p=valgrind.git;a=blob;f=drd/tests/rwlock_race.c;hb=3b3d0d2a25f31f6a7f94f4deccadefoa2ca16699

6 https://clang.llvm.org/docs/ThreadSanitizer.html

Neben den vorgestellten Forschungsarbeiten bieten Linux und FreeBSD eingebaute Mechanismen zur Sperrenanalyse: Linux beispielsweise bietet dem Entwickler den sogenannten *lockdep*-Mechanismus[7] als Hilfestellung [Coro6, Par16]. Er soll Situation aufdecken, die möglicherweise zu Verklemmungen führen können [Coro6, Par16]. Sie beobachten jede statisch-allozierte Sperre [Coro6, Par16]. Dynamisch allozierte Sperren werden hingegen auf der Basis der Stelle im Programmcode, an der sie initialisiert werden, als Klasse zusammengefasst [Coro6, Par16]. Da meist alle Instanzen einer Datenstruktur an der gleichen Stelle initialisiert werden, wird so nach der umgebenden Datenstruktur gruppiert [Coro6, Par16]. Bei der Detektion unterscheidet *lockdep* zwischen zwei Ausgangsszenarien: a) Es wird auf die korrekte Reihenfolge der Sperren geachtet [Coro6, Par16]. Das Holen der Sperren muss ebenso bei allen Beobachtungen in der gleichen Reihenfolge geschehen [Coro6, Par16]. Die Freigabe muss in der inversen Reihenfolge erfolgen [Coro6, Par16]. Alle davon abweichenden Beobachtungen erzeugen eine Warnung [Coro6, Par16]. b) Es werden die unterschiedlichen Ausführungskontexte innerhalb eines Betriebssystemkerns berücksichtigt [Coro6, Par16]. Wie in Unterabschnitt 2.2.1.6 geschildert, können diese unterschiedliche Prioritäten haben. Muss zwischen Kontexten unterschiedlicher Priorität synchronisiert werden, so muss gegenseitiger Auschluss zwischen den Prioritätsebenen sichergestellt sein (vgl. Unterabschnitt 2.2.1.6). Andernfalls kann eine Verklemmung auftreten. Daher zeichnet *lockdep* den Ausführungskontext mit auf [Coro6, Par16]. Wird beispielsweise eine Sperre einmalig in einem Unterbrechungskontext geholt und zu einem anderen Zeitpunkt in einem Anwendungskontext, ohne dass hierbei die Unterbrechungen ausgeschaltet werden, so wird eine Warnung generiert [Coro6, Par16]. Der Linux-Kern bietet nun ebenfalls ein eingebautes Werkzeug zur Detektion von Wettlaufsituationen um Speicherzugriffe an[8].

Der Betriebssystemkern von FreeBSD verfügt über einen vergleichbaren Mechanismus genannt *Witness System* [Balo2, MNNW14]. Dieser soll ebenfalls mögliche Verklemmungen aufdecken [Balo2, MNNW14]. Er zeichnet die Reihenfolge des Holens der Sperren auf und meldet Verstöße gegen die bisherigen Beobachtungen [Balo2, MNNW14]. Zusätzlich wird überwacht, dass keine Sperre geholt wird, die passiv wartet sowie im kritischen Abschnitt eine potentiell schlafende Funktion aufgerufen wird, während bereits eine Sperre gehalten wird, die dies verbietet [Balo2, MNNW14]. Anders als in Linux erfolgt die Gruppierung in Klassen hier anhand eines vom Entwickler gewählten Freitextnamens während der Initialisierung [Balo2, MNNW14].

Komplementär zu den bisher dargestellten Ansätzen zur Laufzeitanalyse ist die Arbeit von Jeong et al. Ihr Ansatz, genannt *Razzer*, findet Wettlaufsituationen bei Speicherzugriffen im Linux-Kern [JKS+19]. Sie konzentrieren sich vornehmlich auf das exakte Provozieren von solchen Fehlern [JKS+19]. In der ersten Stufe werden Kandidaten für potentielle Wettlaufsituationen mittels statischer Code-Analyse gefunden [JKS+19]. Hierbei führen sie die

7 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/locking/lockdep-design.rst?h=v5.6

8 https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html

Alias-Analyse lediglich auf definierten Teilmengen des Quellcodes durch, berücksichtigen dabei die Synchronisationsprimitiven jedoch nicht [JKS+19]. Anschließend versuchen Jeong et al. die potentiellen Wettlaufsituationen in einer virtuellen Maschine mittels der Technik „coverage-guided fuzzing“ zu provozieren [JKS+19]. Nur Kandidaten, für die sie zeigen können, dass sie auch tatsächlich realisierbar sind, werden als problematisch markiert [JKS+19]. Einen verwandten Ansatz verfolgen Xu et al. mit ihrer Arbeit *KRACE* [XKZK20] bzw. Fonseca et al. mit *SKI* [FRB14]. Hervorzuheben ist, dass Xu et al. detailliertere Gedanken über die Integration der diversen Sperrenmechanismen des Linux-Kerns in ihren Ansatz machen [XKZK20]. Diese Arbeiten gehört sowohl zu dem hier geschilderten Themenbereich als auch zu dem Bereich aus Abschnitt 4.1, da manche Schritte offline durchgeführt werden.

4.3 NACHTRÄGLICHE ANALYSE

Dieser Abschnitt lenkt den Blick auf Arbeiten, die sich mit der nachträglichen Analyse beschäftigen: Bei dieser Kategorie von Ansätzen werden zur Laufzeit lediglich die erforderlichen Informationen aufgezeichnet. Die eigentliche Analyse findet nach der Ausführung statt. Dadurch sind diese Ansätze weniger invasiv bezüglich der Laufzeit. Gleichzeitig geben sie mehr Raum, um nachträglich zeitintensive Analysen auf den Daten durchzuführen. Zur Instrumentierung stehen verschiedene Möglichkeiten zur Verfügung: Das Werkzeug *Pin* [LCM+05] beispielsweise erlaubt die dynamische Instrumentierung beliebiger Anwendungen im Binärformat, wenngleich sie selbst eine Laufzeitanalyse durchführen – ähnlich zu *Valgrind* [NS07, SNW08]. Das Werkzeug *Tralfamadore* hingegen führt das zu untersuchende System in einer virtuellen Maschine aus und zeichnet die Informationen mittels einer modifizierten Version von *QEMU* [Bel05] auf [LCH+12]. Die so gewonnenen Informationen können beispielsweise zum Auffinden von Schadsoftware [LSC97] oder aber zur Untersuchung der Prozessablaufsteuerung von Echzeitfäden in Linux [ÅNK11] genutzt werden. Auf der anderen Seite können aber auch aus den Aufzeichnungen Erkenntnisse zur Unterstützung der Entwickler erzeugt werden, um z. B. Einblicke in die Aufrufhierarchie oder aber den Datenfluss innerhalb des Kerns zu erhalten [LCH+12].

KernelStrider ist ebenfalls ein Werkzeug zum Auffinden von Wettlaufsituationen zwischen Speicherzugriffen [Sha19]. Im Gegensatz zu den vorgenannten Werkzeugen aus Abschnitt 4.2 findet die Analyse im Anschluss an das Erheben der Daten statt [Sha19]. Dabei wird die Auswertung mittels *ThreadSanitizer* [SI09, SPIV12] durchgeführt. *KernelStrider* sorgt lediglich für Aufzeichnen der Daten und die Umwandlung in das passende Eingabeformat [Sha19]. Je nach Konfiguration können auch hier Sperren-Operationen berücksichtigt werden [Sha19].

Matni et al. beschreiben in ihrer Arbeit einen generischen Ansatz zur nachträglichen Analyse von Betriebssystemverhalten [Mat11]. Hierzu nut-

zen sie *LTTng* (*Linux Trace Toolkit next generation*)[9] zur Erhebung der Daten [Mat11]. In den Daten suchen sie mittels Zustandsautomaten nach Fehlermustern [Mat11]. Sie beschreiben u.a. einen Automaten zur Detektion von Verklemmungen zwischen Ausführungskontexten im Linux-Betriebssystemkern [Mat11].

4.4 ZUSAMMENFASSUNG

Der Großteil der hier vorgestellten Arbeiten lassen sich nach ihren Zielen unabhängig von dem grundlegenden Vorgehen, statische , Laufzeit- oder expost Analyse, im Wesentlichen in drei Kategorien einteilen:

- Das Auffinden von Wettlaufsituationen zwischen Speicherzugriffen [ECH+01, EA03, LPH+07, VAR+16, BLCH19, SBN+97, NS07, EMBO10, JYX+16, CBJ+19, Sha19, XKZK20],
- die Detektion von Verklemmungen [EA03, BV04, NS07, AS06, JTZC08, Cor06, MNNW14, Mat11] und
- andere Probleme bei der Nutzung von Sperren, wie z. B. falsche Verwendung der Programmierschnittstelle oder aber das Aufdecken von Flaschenhälsen beim Eintritt in einen kritischen Abschnitt [BV04, BR02, Lin04, NS07, HCC+12, BH00, CS12].

Natürlich decken einige Arbeiten mehr als eine Kategorie ab, wie z. B. die Arbeit von Engler et al. [EA03] oder von Nethercote at al. [NS07]. Dennoch lassen die bisherigen Arbeiten einen ganzheitlichen Ansatz vermissen, der dem Entwickler sowohl bei der Entwicklung als auch danach eine Hilfestellung bietet. Die Arbeiten setzen eher nach der Entwicklung an, indem sie den existierenden Programmcode auf Fehler untersuchen.

Der Großteil der Arbeiten befasst sich mit der Detektion von Wettlaufsituationen bei Speicherzugriffen. Dabei liegt jeder Arbeit in irgendeiner Form der *LockSet*-Algorithmus zu Grunde. Er wird genutzt, um den Moment zu erfassen, wenn die Sperren-Menge über alle bisher beobachteten Speicherzugriffe leer ist und somit eine Fehlermeldung zu generieren. Dennoch wird er nirgends genutzt, um Sperren-Mengen für einzelne Datenstrukturen abzuleiten. Lediglich Engler et al. skizzieren, wie ihr Ansatz eingesetzt werden könnte, um festzustellen, ob eine Variable immer von einer bestimmten Sperre geschützt wird [ECH+01].

Abschließend ist festzuhalten, dass die Arbeiten weder die eingangs dieses Kapitels dargelegten Punkte erfüllen noch die Kernanforderungen, das Ableiten der Sperren-Mengen, umfänglich umgesetzten. Daher wird in dieser Arbeit ein Ansatz vorgestellt, um diese Lücke zu schließen. Gleichzeitig greift der in Kapitel 5 und Kapitel 6 vorgestellte LockDoc-Ansatz Ideen existierender Arbeiten auf: Hierzu zählt zunächst einmal die grundlegende Idee einer nachträglichen Analyse (vgl. Kapitel 6). Dies beinhaltet die aufzeichnungsbasierte Analyse wie sie z. B. Matni et al. [Mat11] durchführen.

9 https://lttng.org/

Darüber hinaus setzt LockDoc, analog zu der Arbeit von beispielsweise Lefebvre etl al. [LCH+12], auf eine virtuelle Ausführungsumgebung.

5 ANSATZ

Inhalt

In Kapitel 3 wurde bereits die Bedeutung von Sperren im Betriebssystemkontext herausgestellt, die auch die Analyse der aktuellen Situation umfasste. Daraus resultierend wurden drei wichtige Anforderungen im Bezug auf die Sperren-Dokumentation abgeleitet [LSBS19]: 1) Das Überprüfen von existierender Sperren-Dokumentation hinsichtlich ihrer Korrektheit. 2) Das automatische Generieren einer neuen Sperren-Dokumentation, die einer eindeutigen Syntax folgt. 3) Das Überprüfen des existierenden Quellcodes hinsichtlich der Einhaltung der neuen Sperren-Dokumentation.

Kapitel 4 zeigte bereits, dass bisher keine Arbeit bekannt ist, die solch einen ganzheitlichen Ansatz verfolgt. Ebenfalls in Kapitel 4 wurde als Grundlage für alle Punkte das Bestimmen der Sperren-Mengen für Datenstrukturen genannt. Dieser Punkt wird von den existierenden Arbeiten nur unzureichend abgedeckt. Es obliegt nunmehr dieser Arbeit einen Ansatz vorzustellen, um die Lücke zu füllen. Daher wird im folgenden Kapitel das Grundkonzept für einen aufzeichnungsbasierten Ansatz zur Sperren-Analyse in Betriebssystemen vorgestellt: Bevor in Kapitel 6 der vollständige Arbeitsablauf vorgestellt wird, soll hier zunächst die Grundidee hinter LockDoc vorgestellt werden. Zu diesem Zweck wird in Abschnitt 5.1 zunächst erläutert, wieso ein aufzeichnungsbasierter Ansatz verwendet wird, und, wie dieser aussieht. In Abschnitt 5.2 werden die Grundbegriffe eingeführt sowie die Vorgehensweise anhand eines einfachen Anwendungsbeispiels erläutert.

5.1 AUFZEICHNUNG

Dieser Abschnitt befasst sich mit dem aufzeichnungsbasierten Ansatz. Unterabschnitt 5.1.1 beschäftigt sich mit den aufgezeichneten Daten sowie der Art und Weise der Aufzeichnung. Dabei findet eine Einordnung in die verwandte Arbeiten aus Kapitel 4 statt. Unterabschnitt 5.1.2 widmet sich dediziert der auszuführenden Arbeitslast. Dies umfasst die Analyse einer existierenden Arbeitslast, eine Analyse von verwandten Arbeiten zu dem Thema

sowie ein Konzept zur Verbesserung der Arbeitslast bezüglich der Codeabdeckung.

5.1.1 *Methodik*

Der vorangegangene Abschnitt 4.1 zeigte bereits die Limitierungen der statischen Analyse auf. Dazu zählt insbesondere die Alias-Analyse von Zeigern. Dies fällt besonders ins Gewicht, da gängige Betriebssysteme, wie z. B. Linux oder FreeBSD, in der Programmiersprache *C* geschrieben sind und Zeiger einsetzen: Sie kommen u.a. zur Implementation von Polymorphie zum Einsatz, wie z. B. bei der Anbindung eines Geräte- oder Dateisystemtreibers [Lov10, BC05, Pro21]. Zeiger werden aber auch zur Realisierung von Datenstrukturen mit generischem Inhalt verwendet, wie z. B. einer doppelt verketteten Liste. Das Beispiel aus Abbildung 4.1 für eine verkettete Liste aus dem Linux-Kern machte dies bereits deutlich. Wenngleich Vojdani et al. mit ihrer Arbeit zeigten, dass sich genau dieser Fall auch mittels statischer Analyse lösen lässt, so ist doch ein Ansatz wünschenswert, der nach Möglichkeit solche Fälle im Allgemeinen abdeckt. Daher wird in dieser Arbeit ein aufzeichnungsbasierter Ansatz untersucht. Dieser hat zum Ziel losgelöst vom Programmcode die tatsächlich stattfindenden Zugriffe auf ein Objekt in Form von Speicherzugriffen aufzuzeichnen. Zu diesem Zweck kann z. B. jeder Speicherzugriff instrumentiert werden. Hierfür bieten sich Ansätze wie der von Serebryany et al. an, in dem beim Übersetzen der erzeugte Maschinencode instrumentiert wird [SPIV12]. Da so jeder Speicherzugriff aufgezeichnet wird, müssten die relevanten Speicherzugriffe in einem nachgelagerten Schritt gefiltert werden. Soll sich die Instrumentierung jedoch nur auf die nötigsten Stellen beschränken, so muss beim Übersetzen feststehen, wo relevante Zugriffe stattfinden. Da dem Übersetzer jedoch nur die Analyse des Hochsprachencodes bleibt, werden Szenarien, wie die in Abbildung 4.1 mutmaßlich nicht abdeckt. Dieser Ansatz scheint daher nicht vielversprechend. Aus diesem Grund wird das Zielbetriebssystem bei dem LockDoc-Ansatz in einer virtuellen Umgebung ausgeführt und hier die Speicherzugriffe aufgezeichnet.

Ein ähnliches Vorgehen wählen Lefebvre et al. mit *Tralfamadore* [LCH+12] sowie Jeong et al. mit ihrer Arbeit *Razzer* [JKS+19]. In beiden Fällen werden die benötigten Information in dem virtuellen Prozessor aufgezeichnet. Auf diese Weise werden ebenfalls alle Zugriffe abgefangen. Um auch hier nur die relevanten Speicherzugriffe aufzuzeichnen, ist es sinnvoll, die Aufzeichnung auf Speicherzugriffe, die zu Objekten von relevanten Datenstrukturen gehören, zu beschränken. Relevant sind in diesem Kontext alle Datenstrukturen, die zu dem untersuchten Teilbereich des Betriebssystems gehören. Für LockDoc muss das zu untersuchende Betriebssystem im Betrieb daher mitteilen, welche Objekte (und damit Speicherbereiche) für welche Dauer von Interesse sind. Die virtuelle Ausführungsumgebung nimmt die Information entgegen und beobachtet für den entsprechenden Zeitraum den Speicherbereich auf Zugriffe. Da die Sperren-Mengen in dieser Arbeit ein wichtiger

Bestandteil sind, müssen die Operationen auf den Sperren ebenfalls nach außen kommuniziert und damit aufgezeichnet werden. Hierbei werden die Operationen auf allen Sperren aufgezeichnet, da im vor hinein nicht klar ist, welche Sperren relevant sein könnten. In Summe werden somit die folgenden sechs Ereignisse aufgezeichnet: Das Belegen und Freigeben von allen Sperren, die Lebensdauer (das Allozieren und Freigeben) eines Objektes einer Datenstruktur sowie das Lesen und Schreiben der betrachteten Speicherbereiche. Eine virtuelle Ausführungsumgebung, die das Abfangen der Kommunikation aus dem Gastsystem heraus erlaubt, ist das Werkzeug FAIL* [SHD+15]. Dies wird auch im Rahmen dieser Arbeit eingesetzt. Grundsätzlich können mit dem oben beschriebenen Vorgehen alle Instanzen aller Datenstrukturen eines Betriebssystems beobachtet werden. Da dies jedoch einen erheblichen Mehraufwand bedeutet, beschränkt diese Arbeit auf relevante Datenstrukturen. Diese hängen von dem zu untersuchenden Teil des Betriebssystems ab. In der vorliegenden Arbeit wird sich in beiden Fallstudien auf das Dateisubsystem, das sog. *Virtual File System* (VFS), beschränkt.

Genauere Informationen zu der Ausführungsumgebung finden sich in Kapitel 6.

5.1.2 *Arbeitslast*

Nachdem bisher die Art der Aufzeichnung und die Informationen selbst erläutert wurden, stellt sich abschließend noch die Frage, wie die Speicherzugriffe Zustandekommen: Da die Zugriffe über den Programmcode verteilt sind, müssen verschiedene Teile des Programmcodes ausgeführt werden. Idealerweise wird jeder Pfad durch den Code nur einmal abgelaufen, so dass jeder Speicherzugriff genau einmal stattfindet. Dies würde dafür sorgen, dass die Zugriffe bei einer späteren Analyse gleich gewichtet werden. Da die Ausführungspfade jedoch möglicherweise von einander abhängen, muss die Anforderung relaxiert werden: Es soll jeder Ausführungspfad mindestens einmal ausgeführt werden. Insgesamt soll die Abdeckung des Programmcodes maximiert werden, um möglichst alle Speicherzugriffe einmal auszuführen. Über die Systemaufrufschnittstelle lässt sich das Betriebssystem beispielsweise veranlassen, die entsprechenden Pfade im Programmcode auszuführen. Allerdings hängen die verschiedenen Systemaufrufe voneinander ab. Die Extraktion solcher Abhängigkeiten ist daher Gegenstand aktueller Forschung [PAJ18]. Die Systemaufrufschnittstelle wiederum wird durch ein oder mehrere Anwendungsprogramme verwendet. Somit obliegt es letztlich den Anwendungsprogrammen für die relevanten Speicherzugriffe zu sorgen. Die Sammlung aus Anwendungsprogrammen wird unter dem Begriff Arbeitslast zusammengefasst. Für die Fallstudien in dieser Arbeit wird das sogenannte *Linux Test Project* [Lar02] (LTP) als Arbeitslast verwendet. Das Ziel des Projektes wird auf dessen Webseite dabei wie folgt beschrieben: „validate the reliability, robustness, and stability of Linux“ [H+]. Es besteht dabei aus verschiedenen individuellen Tests, die einzelne Funktionen oder kleinere Gruppen von Funktionen überprüfen [LTS20]. Die Tests werden in sog. Testsuites gruppiert [LTS20]. Jede Suite zielt dabei auf ein bestimmtes Subsystem, wie z. B. auf das VFS, das Speichersubsystem oder einzelne Kern-

Testsuite	# Tests	Basisblöcke	(%)
dio	30	7.978	11,23
fcntl-locktests	1	4.060	5,71
filecaps	1	2.687	3,78
fs	65	17.123	24,10
fs_bind	1	7.644	10,76
fs_ext4	4	12.734	17,92
fs_perms_simple	18	5.030	7,08
fs_readonly	55	12.831	18,06
fsx	1	6.837	9,62
syscalls	1.183	24.314	34,22
Total	1.359	26.258	36,95

Tabelle 5.1: Basisblockabdeckung für das VFS-Subsystem der untersuchten LTP-Testsuiten. Der untersuchte Linux-Kern 5.4 beinhaltet 300.860 Basisblöcke, davon entfallen 71.061 Basisblöcke auf das VFS-Subsystem. (Nach einer Vorlage von Lochmann et al. [LTS20])

Funktionen, wie z.B die Interprozesskommunikation [LTS20]. Dabei reicht die Spannweite der Tests von Stresstests bis zu Regressionstests [LTS20]. Da diese Tests alle manuell erzeugt wurden, können sie nur einen begrenzten Bereich des Parameterraums für die Systemaufrufe abdecken [LTS20]. Dies wiederum resultiert in einer limitierten Codeabdeckung [LTS20]. An dieser stellt sich die Frage, welche Codeabdeckung durch das *Linux Test Project* überhaupt erzielt werden kann. Für den Begriff der Codeabdeckung gibt es in der Literatur verschiedene Metriken, wie z. B. Pfad-, Zweig- oder Zeilenabdeckung [ZHM97, MSB11]. Für die weiteren Analyse wird jedoch die Basisblockabdeckung genutzt, da sie die genauer den Anteil des echt ausgeführten Codes ausdrückt [LTS20]. Eine Zeile Programmcode kann beispielsweise zu mehreren Basisblöcken übersetzt werden [LTS20]. Daraus folgt, dass eine ausgeführte Zeile Programmcode nicht notwendigerweise heißt, dass alle zugehörigen Basisblöcke ausgeführt wurden [LTS20]. Wenn auf der anderen Seite alle Basisblöcke abgedeckt werden, wurden ebenfalls alle erreichbaren Codezeilen ausgeführt [LTS20].

Im Folgenden wird nun die Basisblockabdeckung für alle zum VFS verwandten Testsuites bestimmt. Es wurde die Abdeckung für jeden einzelnen Test aus den folgenden Testsuites ermittelt[1]: *dio*, *fcntl-locktests*, *filecaps*, *fs*, *fs_bind*, *fs_ext4*, *fs_perms_simple*, *fs_readonly*, *fsx*, und *syscalls*. Tabelle 5.1 stellt die Basisblockabdeckung für jede Testsuite dar. Die beiden größten Bei-

1 Es wurde die Version *20190115* aus dem LTP-Repository verwendet.

träge leisten hierbei die Testsuiten *fs* und *syscalls* mit 24,1 % bzw. 34,22 % Basisblockabdeckung. Insgesamt lässt sich jedoch nur eine Basisblockabdeckung von 36,95 % erzielen. Daran wird ersichtlich, dass es Raum für Verbesserungen gibt.

5.1.2.1 *Verwandte Arbeiten*

Die Idee, die Codeabdeckung von Software im Allgemeinen und von Testsuites im Konkreten zu untersuchen, ist nicht neu. In diesem Bereich gibt es bereits verschiedene Arbeiten: Larson et al. untersuchen in ihrer Arbeit die Codeabdeckung im Linux-Kern 2.5 von verschiedener Anwendungssoftware [LHRF03]. Ihre Arbeit evaluieren sie u.a., in dem sie die Codeabdeckung von LTP ermitteln [LHRF03]. Larson et al. messen eine Abdeckung von 35 % der Codezeilen im Kern [LHRF03]. Darüber hinaus stellen sie einen Ansatz zur Generierung von HTML-Berichten vor, die die erhobenen Daten, wie die Codeabdeckung, in den Programmcode einbetten und graphisch aufbereiten [LHRF03]. Iyer analysiert ebenfalls die Codeabdeckung von *Linux Test Project*[2] [Lar02] mittels *GCOV*[3] [Iye02]. Sie messen u.a. knapp 30 % Codeabdeckung für das Verzeichnis `fs/` im Linux-Kern 2.4 [Iye02]. Yoshioka entwickelte mit *crackerjack* ein Rahmenwerk für Regressionstests für den Linux-Kern [Yos07]. Dies schließt das Werkzeug *btrax* zur Bestimmung der Funktions-, Pfad-, und Zustandsabdeckung ein [Yos07]. Der Vorteil von *crackerjack* gegenüber Testsuites wie z. B. LTP liegt in dem Vergleich von der Ausgabe von zwei Versionen der gleichen Software und damit einhergehend der Detektion von geändertem Verhalten [Yos07]. Überdies vermessen sie noch eine Teilmenge der LTP-Tests nach der vorgenannten Metrik [Yos07]. Hierbei fällt auf, dass z. B. gerade mal 10 % der untersuchten Programme eine Pfadabdeckung von über 50 % erreichen [Yos07].

Claudi et al. stellen mit ihrer Arbeit *Lachesis* ebenfalls eine Testsuite vor [CD11]. Diese richtet sich jedoch primär an die Echtzeiterweiterungen für Linux [CD11]. Die ersten Ansätze zur automatischen Parameterraumexploration der Systemaufrufschnittstelle mittels *fuzzing* gehen zurück auf Tin Le (*tsys* [Le91]) und Jones et al. (*trinity* [J+06, Mic13]). Der Ansatz von Jones et al. widmete sich dabei als eines der ersten Werkzeuge dem Linux-Kern [J+06, Mic13]. Moderne Ansätze integrieren eine Rückkopplung in den *fuzzing*-Prozess, um die erreichte Codeabdeckung zu berücksichtigen. Ein prominenter Vertreter ist der von Dmitriy Vyukov entwickelte Ansatz *syzkaller* [Vyu15, Dav16]. Abseits davon gibt es noch weitere Arbeiten, die sich diesem Thema widmen: Nossum und Casanova portieren den aus dem Bereich der Anwendungssoftware bekannten Ansatz *american fuzzing loop* auf den Linux-Kern [Jak16, NC16]. Sie wenden im Weiteren ihren Ansatz auf verschiedene Dateisysteme, wie z. B. *ext4*, an, um Fehler zu finden [Jak16, NC16].

Andere Autoren verfeinern ihre Ansätze z. B. mit statischer Code-Analyse: Corina et al. mit *DIFUZE* zielen auf das Finden von Fehlern in Gerätetreibern

2 https://github.com/linux-test-project/ltp/
3 https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.htmlo

```
1  int main(void)
2  {
3    syscall(__NR_mmap, 0x1ffff000, 0x1000, 0, 0x32, -1, 0);
4    syscall(__NR_mmap, 0x20000000, 0x1000000, 7, 0x32, -1, 0);
5    syscall(__NR_mmap, 0x21000000, 0x1000, 0, 0x32, -1, 0);
6
7    *(uint32_t*)0x20002480 = 0x20000340;
8    memcpy((void*)0x20000340, "\x12", 1);
9    *(uint32_t*)0x20002484 = 1;
10   *(uint32_t*)0x20002488 = 0;
11   syz_read_part_table(0, 1, 0x20002480);
12   return 0;
13 }
```

Listing 5.1: Ein Auszug eines von *syzkaller* zufällig generierten Programms [Vyu15]. (Von Lochmann et al. [LTS20])

ab [CMS+17]. Dabei analysieren sie vorab den Quellcode, um den Parameterbereich für die zu untersuchenden Gerätetreiber zu schärfen [CMS+17]. Im Gegensatz dazu entwerfen Schumilo et al. den Betriebssystem-unabhängigen Ansatz *kernel AFL* [SAG+17]. Dieser nutzt die Funktion *processor trace* von Intel-Prozessoren zur Bestimmung der Codeabdeckung [SAG+17]. Shit et al. zeigen die Herausforderungen bei der Anwendung von *fuzzing*-Techniken mittels *syzkaller* auf Enterprise-Versionen des Linux-Kerns auf [SWF+19].

5.1.2.2 *Verbesserung der Arbeitslast*

Wie im vorangegangenen Abschnitt ausgeführt, existieren bereits *fuzzing*-Ansätze, die die Codeabdeckung mit einbeziehen. Vielmehr noch: Ihr Ziel ist es die Codeabdeckung zu maximieren. Ein prominenter Vertreter ist der von Dmitry Vyukov entwickelte *syzkaller*-Ansatz [Vyu15]. Dabei wird mittels zufällig erzeugten Anwendungsprogrammen der Linux-Kern über die Systemaufrufschnittstelle getestet. Ein Beispiel dafür ist in Listing 5.1 gegeben. Jedes Programm wird mit jeder Iteration mit dem Ziel modifiziert bzw. erweitert, einen Fehler im Kern auszulösen. Tritt ein Fehler auf, wird das verwendete Programm automatisiert so weit minimiert, dass nur noch die Fehler auslösenden Teile übrigbleiben. Terminiert hingegen ein Programm korrekt, wird überprüft, ob es mindestens einen neuen Basisblock abdeckt hat. Ist dies der Fall, so wird es im Korpus abgespeichert. Diese Vorgehensweise wurde nun so modifiziert, dass 1) Programme, die Fehler auslösen, ignoriert werden und 2) nur Programme abgespeichert werden, die neue Basisblöcke im VFS-Subsystem abdecken [LTS20]. Die so erzeugten Programme sollen die Codeabdeckung steigern. Dies wiederum deckt mehr Speicherzugriffe und Sperren-Operationen ab, die von LockDoc für die Analyse genutzt werden können. Die so generierten Programme können nicht (direkt) als Regressionstests für den Linux-Kern eingesetzt werden, da sie keine explizite Ausgabe tätigen, die einen Erfolg vermelden könnte [LTS20].

5.2 ANALYSE

In diesem Abschnitt wird die Verarbeitung und Auswertung der zuvor aufgezeichneten Daten beschrieben. Bevor jedoch im Detail auf die weiteren Schritte eingegangen wird, sollen zunächst die wichtigsten Begriffe und deren Beziehungen zueinander vorgestellt werden. Wie bereits in Unterabschnitt 5.1.1 geschildert wurde, ist ein wesentliches Ziel dieser Arbeit die Bestimmung von Sperren-Mengen für jedes Element einer Datenstruktur in dem zu untersuchenden Betriebssystem. Dies lässt sich wie folgt beschreiben: Die Menge S stellt alle Elemente aller Datenstrukturen in einem Betriebssystem dar – vgl. Gleichung 5.1. Gleichzeitig beschreibt die Menge N alle in dem betreffenden Betriebssystem vorhandenen Sperren – vgl. Gleichung 5.2.

$$S := \{s : \text{Element einer Datenstruktur im Betriebssystem}\} \tag{5.1}$$

$$N := \{n : \text{eine Sperre in dem Betriebssystem}\} \tag{5.2}$$

$$M := \mathcal{P}(N) = \{U | U \subseteq N\} \tag{5.3}$$

Da eine Sperren-Menge für ein Element keine, eine oder mehrere Sperren enthalten kann, wird von der Menge N die Potenzmenge gebildet, so dass sich die Menge M ergibt – vgl. Gleichung 5.3. Diese enthält alle Teilmengen aus der Menge N. Die Idee hinter dem LockDoc-Ansatz ist es, nun eine Abbildung von S nach M durchzuführen, so dass jedem Element eine Sperren-Menge $L \in M$ zugeordnet wird. Da die Abbildung nach M erfolgt, ist aufgrund der Definition der Potenzmenge auch die leere Menge enthalten. Dies bedeutet, dass für einen Zugriff auf das jeweilige Element keine Sperren erforderlich sind. Somit ist ersichtlich, welche Sperren für einen Zugriff erforderlich sind. Diese Definition ist verwandt zu der Definition einer Sperren-Menge von Savage et al. [SBN+97].

$$S' := \{s : \text{Element einer beobachteten Datenstruktur}\} \tag{5.4}$$

$$N' := \{n : \text{eine aufgezeichnete Sperre in dem Betriebssystem}\} \tag{5.5}$$

$$M' := \mathcal{P}(N') = \{U | U \subseteq N'\} \tag{5.6}$$

Allerdings wird für die Analyse eine Aufzeichnung und keine Analyse des Quellcodes herangezogen. Daher werden nicht die Elemente aller Datenstrukturen betrachtet, sondern die Elemente aller beobachteten Datenstrukturen, wie in Unterabschnitt 5.1.1 erklärt wurde. Somit ergibt sich eine neue Menge S' , die alle Elemente der beobachteten Datenstrukturen umfasst – vgl. Gleichung 5.4. Daher gilt: $S' \subseteq S$. Aus dem selben Grund bedarf auch die Menge aller Sperren einer Konkretisierung: Die Menge N' umfasst alle aufgezeichneten Sperren des zu untersuchenden Betriebssystems – vgl. Gleichung 5.6. Entsprechend der Definition der Menge M wird die Menge M' als Potenzmenge $\mathcal{P}(N')$ definiert. Analog gilt hier: $N' \subseteq N$. Nach dem Vorgehen aus Unterabschnitt 5.1.1 werden die Operationen aller Sperren eines Betriebssystems aufgezeichnet. Bisher wurde jedoch nicht genauer formuliert, was dies bedeutet. Betrachtet man die Schilderungen in

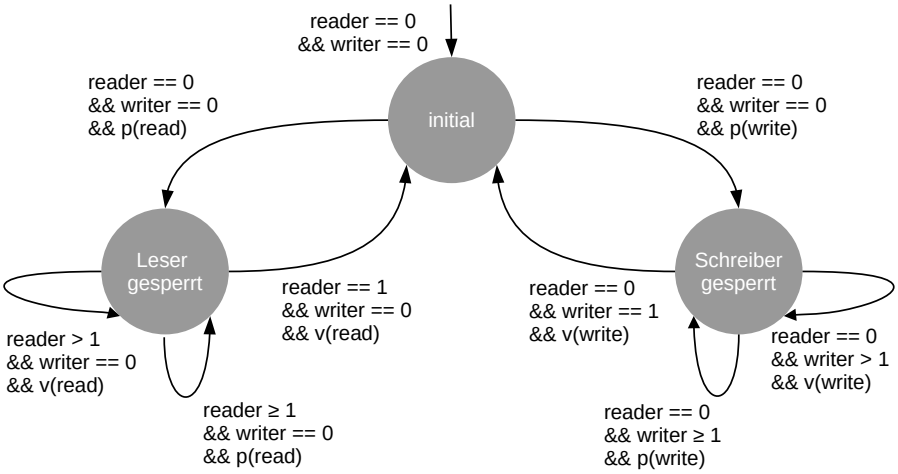


Abbildung 5.1: Sperren-Modell der von LockDoc unterstützten Sperren-Typen. Der linke Teil repräsentiert eine Leser-Sperre, der rechte Teil eine Schreiber-Sperre. Diese unterstützt das rekursive Belegen der Sperre. Jeder Sperren-Typ, der eine oder beide Konzepte realisiert, wird von LockDoc unterstützt.

Unterabschnitt 2.2.1 sowie in der Literatur [Lov10, MNNW14] zu verschiedenen Betriebssystemen so zeigt sich eine ganze Bandbreite an Sperren-Typen. LockDoc unterstützt hiervon Synchronisationsmechanismen zur expliziten Synchronisation zur Herstellung von gegenseitigem Ausschluss. Einseitige Synchronisation, wie bei Speicherbarrieren [Lov10], oder implizite Synchronisation, wie dies z. B. bei Monitoren (vgl. Unterabschnitt 2.2.1) der Fall ist, werden nicht unterstützt. Für LockDoc wird zwischen zwei Formen des gegenseitigen Ausschlusses unterschieden: geteilter und exklusiver gegenseitiger Ausschluss. Ersterer kommt beispielsweise bei dem Leser-Schreiber-Problem (vgl. Unterabschnitt 2.2.1.1) vor. Hier herrscht gegenseitiger Ausschluss zwischen den Lesern und den Schreibern. Exklusiver gegenseitiger Ausschluss lässt sich mit klassischen Sperren, wie z. B. dem binären Semaphor oder einer Mutex (vgl. Unterabschnitt 2.2.1.2), realisieren. Beide Varianten sind in LockDoc abbildbar: Abbildung 5.1 zeigt ein Modell für den generischen Sperren-Typ, wie er von LockDoc unterstützt wird. Es stellt eine Leser-Schreiber-Sperre dar. Diese lässt sich in eine Leser-Sperre (linker Teil) und eine Schreiber-Sperre (rechte Teil) aufteilen. Unterstützt ein bestimmter Sperren-Typ das rekursive Belegen der Schreiber-Sperre, so ist dies auch in LockDoc abgebildet – siehe rechter Teil in Abbildung 5.1. Jeder Sperren-Typ eines Betriebssystems, der einem oder beiden Konzepten folgt, lässt sich in LockDoc abbilden. Damit ist insbesondere der Mechanismus *Read-Copy-Update* (vgl. Unterabschnitt 2.2.1.7), wie ihn der Linux-Kernel verwendet, als reine Leser-Sperre abbildbar. Ein ähnliches Modell zur Unterscheidung zwischen Leser- und Schreiber-Sperren haben Savage et al. in ihrer Arbeit ebenfalls realisiert [SBN+97].

$$S'' := \{s : \text{ein zugegriffenes Element einer beobachteten Datenstruktur}\} \tag{5.7}$$

Aufgrund des aufzeichnungsbasierten Ansatzes ist jedoch nicht garantiert, dass auf jedes Element im Verlauf der Ausführung zugegriffen wird – vgl. Unterabschnitt 5.1.2. Daher kann nur den Elementen einer Datenstruktur eine Sperren-Menge zugeordnet werden, für die auch tatsächlich Zugriffe aufgezeichnet wurden. Dementsprechend muss die Menge S' weiter eingeschränkt werden: Die Menge S'' umfasst alle zugegriffenen Elemente einer beobachteten Datenstruktur – vgl. Gleichung 5.7. Es gilt: $S'' \subseteq S' \subseteq S$. Aus diesen Schilderungen folgt, dass mit Hilfe von LockDoc eine Abbildung von S'' nach N' vorgenommen wird. Es wird jedem zugegriffenen Element einer beobachteten Datenstruktur eine Sperren-Menge $L \in M'$ zugeordnet.

Aus der Definition einer Sperren-Menge ergibt sich, dass nur die beteiligten Sperren spezifiziert werden. Die Reihenfolge, in der sie belegt werden sollen, ist nicht ersichtlich – diese ist jedoch wichtig, um z. B. Verklemmungen zu vermeiden. Daher werden zusätzlichen die Begriffe der Sperren-Regel oder Sperren-Reihenfolge definiert. Eine Sperren-Regel ist eine Abfolge von Sperren verbunden mit dem Pfeil-Operator, die die zeitliche Abfolge des Belegens ausdrückt. Die Sperren-Regel $a \rightarrow b$ beispielsweise bedeutet, dass die Sperre a vor der Sperre b belegt wurde.

Mit Hilfe dieser Definitionen wird im Folgenden das weitere Vorgehen des LockDoc-Ansatzes erläutert: Dies setzt sich aus 1) der Analyse der Variablenzugriffe in Kombination mit den gehaltenen Sperren (Abschnitt 5.2.2) sowie 2) dem Ableiten der Sperren-Hypothesen und der Auswahl der Gewinner-Hypothese (Abschnitt 5.2.3) zusammen. Abschließend werden in Unterabschnitt 5.2.4 verschiedene Auswahlstrategien vorgestellt.

5.2.1 *Ein Uhrzeitzähler als Anwendungsbeispiel*

Das Listing 5.2 zeigt den Programmcode für einen einfachen Uhrzeitzähler [LSBS19]. Die Uhrzeit wird in diesem Szenario durch die geteilten Variablen `seconds` und `minutes` repräsentiert [LSBS19]. Zum Schutz vor konkurrierenden Zugriffen wird die Variable `seconds` durch die Sperre `sec_lock` abgesichert [LSBS19]. Erreicht die Variable den Wert 60, so wird noch die Sperre `min_lock` geholt, um den Zugriff auf die Variable `minutes` abzusichern [LSBS19]. In einer hypothetischen Aufzeichnung finden sich nun 1000 Ausführungen des gezeigten Programmcodes [LSBS19]. Darüber hinaus enthält die Aufzeichnung aber eine einzelne Ausführung eines weiteren, ähnlichen Code-Fragments [LSBS19]. Hier wurde jedoch vom Entwickler die Sperre `min_lock` vergessen, so dass hier die Möglichkeit von Wettlaufsituationen beim Zugriff auf die Variable `minutes` besteht [LSBS19]. Anhand dieses Beispiels wird bereits eine Grundannahme des LockDoc-Ansatzes deutlich: Der Großteil des zu untersuchenden Programmcodes arbeitet korrekt [LSBS19]. Fehler sind hingegen rar [LSBS19]. Dies geht konform mit der verwandten Grundidee

```
1 lock(&sec_lock); // Transaktion 1 - Start
2 seconds = seconds + 1;
3 if (seconds == 60) {
4     lock(&min_lock); // Transaktion 2 - Start
5     seconds = 0;
6     minutes = minutes + 1;
7     unlock(&min_lock); // Transaktion 2 - Ende
8 }
9 unlock(&sec_lock); // Transaktion 1 - Ende
```

Listing 5.2: Exemplarische Implementierung eines Uhrzeitzählers. Die Variablen `seconds` und `minutes` zählen jeweils die Sekunden bzw. Minuten. Zugriffe darauf werden durch die zugehörigen Sperren `sec_lock` und `min_lock` geschützt. (Aus einem Papier von Lochmann et al. [LSBS19])

hinter der Arbeit von Savage et al.: Sie betiteln ihre Arbeit als „Bugs As Deviant Behavior [...]“ [ECH+01]. Die betriebssystemeigenen Mechanismen, wie eben *lockdep* [Cor06] oder *witness* [MNNW14] gehen ähnlich vor, indem sie auffälliges, abweichendes Verhalten detektieren.

5.2.2 *Analyse der Variablenzugriffe*

Zum besseren Verständnis der folgenden Analyse der Variablenzugriffe (Schritt 1) muss zunächst das Konzept der Transaktionen vorgestellt werden [LSBS19]. Eine Transaktion (txn) $T_{i \in \mathbb{N}}$ beschreibt eine Abfolge von gehaltenen Sperren, wobei mindestens eine Sperre enthalten sein muss. Eine Transaktion startet mit dem Holen einer Sperre und endet mit deren Freigabe [LSBS19]. Die Transaktion T_1 in Listing 5.2 startet beispielsweise in Zeile 1. Sie endet in Zeile 9 mit der Freigabe. Der Transaktion T_1 ist nur die Sperre a zugeordnet. Mit dem Belegen einer weiteren Sperre in Zeile 4 wird eine weitere Transaktion T_2 angelegt. Werden zwei aufeinander folgende Transaktionen T_i und T_{i+1} verschachtelt, wie in Zeile 4, so besteht die Abfolge der Transaktion T_{i+1} aus der Abfolge von T_i plus der eben belegten Sperre. Dementsprechend wird der Transaktion T_2 die Abfolge $a \rightarrow b$ zugeordnet. Auf diese Weise unterscheiden sich zwei aufeinander folgende und verschachtelte Transaktionen T_i und T_{i+1} genau um die eben belegte Sperre.

Alle nicht-beendeten Transaktionen werden mit Hilfe eines Stapels verwaltet. Die oberste Transaktion stellt die gerade aktive Transaktion dar. Ihr werden alle stattfindenden Variablenzugriffe zugeordnet. In dem vorliegenden Beispiel in Listing 5.2 werden alle Variablenzugriffen zwischen Zeile 4 und 7 daher der Transaktion T_2 zugeschlagen [LSBS19]. Die Zugriffe zwischen den Zeilen 1 und 4 bzw. 7 und 9 werden der Transaktion T_1 zugerechnet [LSBS19]. Durch das Konzept der Transaktionen lassen sich nun Variablenzugriffe einer festen Menge an gehaltenen Sperren zuordnen [LSBS19].

Tabelle 5.2 listet die Zugriffe getrennt nach Variable und Zugriffstyp auf. Die Unterscheidung nach Zugriffstyp erfolgt, da für einen lesenden Zugriff

Variable	Zugriffstyp	Beobachtet		Aggregiert		WoR	
		T_1	T_2	T_1	T_2	T_1	T_2
`seconds`	r	2	0	1	0	0	0
	w	1	1	1	1	1	1
`minutes`	r	0	1	0	1	0	0
	w	0	1	0	1	0	1

Tabelle 5.2: Zugriffe auf die Variablen `seconds` und `minutes` gruppiert nach dem Zugriffstyp. Jede Zeile enthält die Anzahl der tatsächlich beobachteten Zugriffe (*Beobachtet*), der aggregierten Zugriffe (*Aggregiert*) sowie die Fälle, wo ein Schreibzugriff einen Lesezugriff verdeckt (*WoR = Write over Read*). (Aus einem Papier von Lochmann et al. entnommen [LSBS19])

möglicherweise eine andere, weniger strengere Sperren-Regel gelten könnte [LSBS19]. Die Variable `seconds` wird in Transaktion T_1 beispielsweise zweimal (Spalte *Beobachtet*) gelesen [LSBS19]. Würde diese Anzahl für die weiteren Schritte verwendet, würde die Transaktion T_2 das Gewicht Zwei erhalten. Da jedoch unklar ist, ob die beobachteten Zugriffe innerhalb der Transaktion T_2 mit der korrekten Sperren-Reihenfolge erfolgt, werden die Zugriffe zusammengefasst (Spalte *Aggregiert*) [LSBS19]. Für jedes Tupel aus Variable, Zugriffstyp und Transaktion wird somit nur die binäre Information, ob ein Zugriff stattgefunden hat, gespeichert [LSBS19]. Somit wird jede Transaktion gleich gewichtet [LSBS19].

Die Werte aus der Spalte *Aggregiert* können nun herangezogen werden, um Sperren-Regeln abzuleiten. In der bisherigen Version des LockDoc-Ansatzes wurde die Informationen noch weiter reduziert: Die lesenden Zugriffe innerhalb einer schreibenden Transaktionen werden ignoriert (Spalte *WoR = Write over Read*) [LSBS19]. Die vorgenannte Annahme zur Differenzierung zwischen lesenden und schreibenden Zugriffen impliziert jedoch, dass es eine Verwandtschaftsbeziehung zwischen den Sperren-Regeln gibt: Eine strengere Sperren-Regel, beispielsweise $a \rightarrow b$, für schreibende Zugriffe enthält mindestens die gleichen Sperren wie eine Regel für lesende Zugriffe, wie z. B. a. Lesende Zugriffe innerhalb einer schreibenden Transaktion stützen somit auch die Regel für einen lesenden Zugriff, hier die Regel a. Das Vernachlässigen dieser Lesezugriffe (Spalte *WoR*) birgt die Gefahr, dass die eigentliche Regel sich nicht durchsetzt. Beide Möglichkeiten werden in Kapitel 7 miteinander verglichen. Da im Folgenden zur Erklärung des Ableiten der Sperren-Regeln das Schreiben der Variable `minutes` betrachtet wird, ist diese Unterscheidung vorerst nicht relevant.

5.2.3 *Ableiten der Sperren-Regeln*

Auf Basis der zuvor ermittelten Zugriffe können nun für Schritt (2) die Hypothesen für die Sperren-Regeln ermittelt und daraus eine Gewinnerhypothese ausgewählt werden. Die Sperren-Regeln werden immer nur für ein Paar aus Zugriffstyp und einer Variable abgeleitet [LSBS19]. Korrelationen zwischen

ID	Sperren-Hypothesen	s_a	s_r
#0	*Keine Sperre nötig*	17	100%
#1	`sec_lock`	17	100%
#2	`sec_lock` → `min_lock`	16	94.12%
#3	`min_lock`	16	94.12%
#4	`min_lock` → `sec_lock`	0	0%

Tabelle 5.3: Mögliche Sperren-Hypothesen für den Schreibzugriff auf die Variable `minutes` sowie den absoluten und relativen Support (s_a und s_r). (Aus einem Papier von Lochmann et al. entnommen [LSBS19])

Zugriffen auf mehrere Variablen innerhalb eines kritischen Abschnitts, wie sie z. B. Lu et al. [LPH+07] finden, werden von dem LockDoc-Ansatz bisher nicht betrachtet. Es werden nur Sperren-Regeln für einzelne Variablen bestimmt.

Das Vorgehen soll nun anhand der Schreibzugriffe auf die Variable `minutes` gezeigt werden. Zunächst werden alle Transaktionen gesucht, in denen die Variable `minutes` geschrieben wird [LSBS19]. Anhand aller hierbei beobachteten Sperren wird zunächst ohne Berücksichtigung der Reihenfolge die Potenzmenge generiert [LSBS19]. Für das vorliegende Beispiel ergeben sich daher vier Mengen: (a) die leere Menge, (b) Sperre `sec_lock`, (c) Sperre `min_lock` sowie (d) die Sperren `sec_lock` und `min_lock`. Die leere Sperren-Menge repräsentiert dabei die Regel „keine Sperren nötig“ [LSBS19]. Für jede dieser Mengen werden nun unter Einbeziehung der Reihenfolge alle möglichen Permutationen gebildet [LSBS19]. Wie in Tabelle 5.3 zu sehen ist, ergeben sich in Summe fünf Hypothesen. Eine Hypothese ist eine noch in dem Auswahlprozess befindliche Sperren-Regel. Aus (d) lassen sich beispielsweise die Hypothesen #2 und #4 ableiten. Für jede der daraus resultierenden Hypothesen wird sowohl der relative als auch der absolute Support berechnet: Der absolute Support s_a bezeichnet alle Variablenzugriffe – gemäß der Metrik aus Abschnitt 5.2.2 –, bei denen die entsprechenden Sperren gehalten waren [LSBS19]. Hierbei spielt es keine Rolle, ob noch weitere Sperren an einem bestimmten Zugriff beteiligt waren [LSBS19]. Entscheidend ist lediglich, dass die Sperren in der betreffenden Reihenfolge gehalten waren [LSBS19]. Für die leere Menge bedeutet dies offensichtlich, dass sie bei jedem Zugriff zutrifft ($s_a = 17$) [LSBS19].

Da die Transaktion T_2 in Listing 5.2 16-mal[4] ausgeführt wird, ergibt sich in Tabelle 5.3 sowohl für Hypothese #2 als auch für Hypothese #3 jeweils ein absoluter Support von 16. Da es noch einen weiteren, inkorrekten Zugriff auf die Variable `minutes` gab, erhält die Hypothese #1 einen absoluten Support von 17 – wie auch Hypothese #0.

Der relative Support s_r einer Hypothese ist der Quotient aus dem absoluten Support der jeweiligen Hypothese und allen beobachteten Zugriffen des jeweiligen Typs – lesend oder schreibend [LSBS19]. Anhand dieser Me-

4 1000 Ausführungen dividiert durch 60 Sekunden.

triken lassen sich die Hypothesen nun, wie bereits in Tabelle 5.3 zu sehen, sortieren.

Im Folgenden soll die Auswahl der Gewinnerhypothese beleuchtet werden. Als erster Ansatz wird der *LockSet*-Algorithmus [SBN+97] betrachtet. Dieser bildet die Schnittmengen über alle Sperren-Mengen aller Zugriffe. Auf den LockDoc-Ansatz übertragen bedeutet dies, dass die Hypothese, die bei jedem Zugriff ($s_r = 1$) zutrifft und die meisten Sperren beinhaltet, gewinnt. Gibt es keine Hypothese, die mindestens eine Sperre enthält, würde die Regel „Keine Sperren nötig“ gewinnen. In dem vorliegenden Beispiel gewinnt nach dem *LockSet*-Algorithmus die Hypothese #1. Führt man sich nochmal das Beispiel aus Abschnitt 5.2.1 vor Augen, so wird klar, dass dies für den Schreibzugriff auf die Variable `minutes` offenkundig die falsche Gewinnerhypothese ist. Die korrekte Gewinnerhypothese wäre jedoch Hypothese #2. Aufgrund des einen fehlerhaften Zugriffs erhält Hypothese #1 jedoch einen relativen Support von 100 %.

Eine weitere Möglichkeit zur Bestimmung der Gewinnerhypothese ist folgender, naiver Ansatz: Als Gewinner wird die Hypothese mit dem größten relativen Support über einem bestimmten Akzeptanzschwellenwert t_{ac}, wie z. B. 90 %, ausgewählt. Gleichzeitig wird die Hypothese „Keine Sperren nötig“ hier gesondert behandelt: Erst ab einer bestimmten Menge an Zugriffen ohne jegliche Sperre wird sie als Gewinner bestimmt. Trotz dieser Sonderbehandlung würde in dem vorliegenden Beispiel erneut die Hypothese #1 gewinnen. Das ist, wie zuvor bereits erläutert, jedoch die falsche Hypothese.

Daher wird für LockDoc eine andere Herangehensweise gewählt: Die Grundannahme, dass der Programmcode meistens korrekt arbeitet, impliziert, dass es vereinzelt Fehler geben kann. Bezogen auf die Hypothesen und die Metriken bedeutet dies wiederum, dass die Gewinnerhypothese nicht zwingend einen relativen Support von 100% hat. Daher wird an die Liste der Hypothesen von unten herangegangen [LSBS19]. Hierzu wird zunächst nur der Teil betrachtet, dessen relativer Support über einem Akzeptanzschwellenwert ($s_r \geq t_{ac}$) liegt [LSBS19]. Aus diesem Teil ($t_{ac} \leq s_r \leq 1$) wird die Hypothese mit dem kleinsten relativen Support gewählt [LSBS19]. Haben mehrere Hypothesen den gleichen relativen Support, wird die Hypothese mit den meisten Sperren gewählt. Ist die Anzahl der Sperren ebenfalls gleich, wird aus den verbleibenden Hypothesen in FIFO-Reihenfolge ausgewählt, da anhand der bisherigen Metriken keine differenzierte Entscheidung getroffen werden kann. Es werden aber alle, verbleibenden Hypothesen als Gewinner markiert. Die leere Sperren-Menge in der Auswahlprozedur als gewöhnliche Hypothese mit eingebunden. Sie wird automatisch ausgewählt, sollte es keine Hypothese oberhalb des Akzeptanzschwellenwerts geben. Dementsprechend gewinnt in dem Beispiel die Hypothese #2, welches auch die korrekte Hypothese darstellt. Als Akzeptanzschwellenwert wird zunächst 90% gewählt [LSBS19]. Hierdurch soll das Grundrauschen in Form von zufällig gehaltenen Sperren gefiltert werden [LSBS19]. Auf den Einfluss des Akzeptanzschwellenwerts auf die Ergebnisse wird in Kapitel 7 genauer eingegangen.

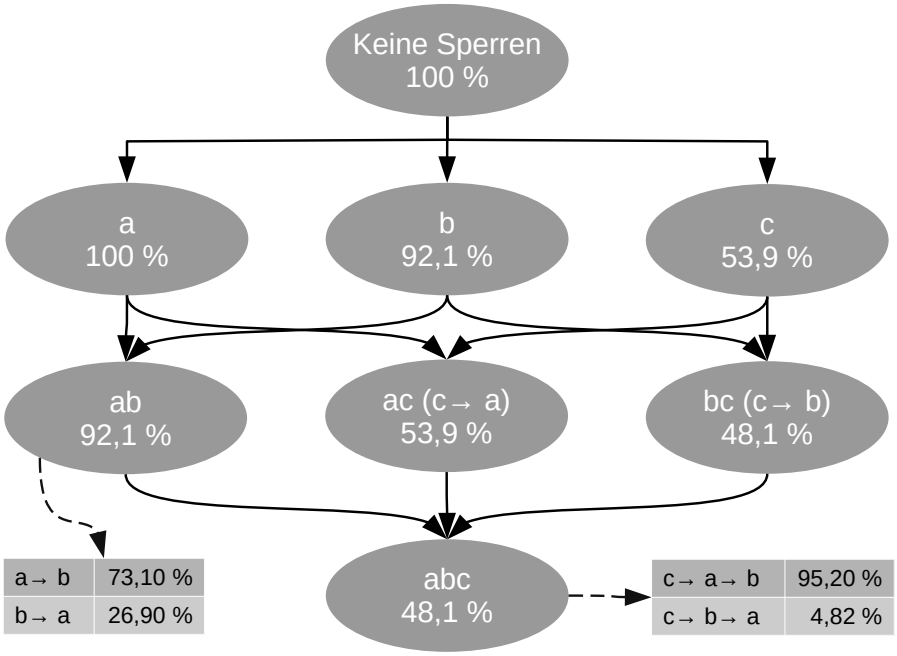


Abbildung 5.2: Exemplarische Darstellung einer Sperren-Mengen als Graph. Zum besseren Verständnis sind nur ein Teil aller beobachteten Sperren-Mengen und Hypothesen dargestellt. Insgesamt sind drei verschiedene Sperren *a*, *b* und *c* zu sehen. Die Knoten *ab* und *abc* wurden zusätzlich mit den dahinter liegenden Hypothesen annotiert (gestrichelter Pfeil).

5.2.4 *Auswahlstrategien*

In Abschnitt 5.2.3 wurde erklärt, wie im Kontext von LockDoc aus der Liste aller Hypothesen eine Gewinnerhypothese ermittelt wird. Hierbei wurde deutlich, dass es nicht offensichtlich ist, wie eine Gewinnerhypothese zu bestimmen ist. Daher soll in diesem Abschnitt ein einheitliche Darstellung verschiedener Auswahlstrategien erfolgen. Dazu werden insgesamt vier verschiedene Auswahlstrategien vorgestellt und in Kapitel 7 miteinander verglichen. Für ein besseres Verständnis soll zunächst eine einheitliche Darstellung der Hypothesen als gerichteter Graph eingeführt werden: Ein gerichteter Graph $G = (V, E)$ beinhaltet dabei alle Hypothesen für ein Tupel aus Datentyp, Element und Zugriffstyp, für die es mindestens eine Beobachtung gibt. Der Ausgangspunkt für jeden Graph ist, wie in Abbildung 5.2 dargestellt, die leere Sperren-Menge: der Knoten v_0. Sie repräsentiert die Regel *„keine Sperren nötig“*. Jeder weitere Knoten $v \in V$ stellt dabei zunächst genau eine Sperren-Menge dar. Hinter einem Knoten v verbergen sich alle Hypothesen, die die Sperren aus der Sperren-Menge L_v [5] beinhalten. Eine gerichtete Kante $e = (v, v') \in E$ verbindet zwei Knoten v und v', wenn die zu v' gehörende Sperren-Menge genau um eins größer ist als die zu v gehörende Menge: $|L_{v'}| = |L_v| + 1$ und $L_v \subset L_{v'}$. Jeder Knoten enthält damit implizit mindestens eine Hypothese, wie z. B. die Knoten *ac* oder *bc* in Abbildung 5.2.

5 Die Menge L_v stellt die Sperren-Menge zu dem Knoten v dar.

Darüber hinaus ergibt sich die Kardinalität der Sperren-Menge, die zum Knoten v gehört, aus der Länge des Weges von v_0 zu v:$|(v_0, \ldots, v)| = |L_v|$. Der Pfad (v_0, b, ab) in Abbildung 5.2 hat beispielsweise die Länge 2. Mit jedem weiteren Knoten auf dem Weg kommt eine weitere Sperre hinzu. Dementsprechend gibt es von v_0 einen Weg der Länge 1 zu jeder Sperren-Menge der Kardinalität 1.

Darüber hinaus ist mit jedem Knoten zusätzlich der relative Support s_r der jeweiligen Sperren-Menge assoziiert. Dieser umfasst den relativen Support von allen dahinter liegenden Hypothesen. Sollte sich hinter einem Knoten mehr als eine Hypothese verbergen, so ist für jede Hypothese der Anteil an dem relativen Support s_r der Sperren-Menge vermerkt. Von beispielsweise 48,10 % der Beobachtungen zu der Sperren-Menge hinter dem Knoten abc entfallen davon 95,20 % auf die Hypothese $c \rightarrow a \rightarrow b$.

Im Folgenden sollen nun die vier Strategien zur Bestimmung der Gewinnerhypothese vorgestellt werden. Alle Strategien starten im Knoten v_0. Sie pessimieren ihr Vorgehen, in dem sie annehmen, dass mehr Sperren besser sind. D.h. die Gewinnerhypothese wird immer die maximal mögliche Anzahl an Sperren enthalten. Außerdem wird bei der Betrachtung eines Knotens immer nur die Hypothese mit dem größten Anteil ausgewählt und für den Vergleich herangezogen, sofern es zu einer Sperren-Menge mehr als eine Hypothese gibt. Für den Knoten ab in Abbildung 5.2 wäre dies beispielsweise die Hypothese $a \rightarrow b$. Ebenso terminieren alle Strategien, sobald es keine Knoten mehr zu untersuchen gibt. Wird jedoch ein neuer bester Knoten gefunden, so werden alle Nachfolgeknoten zur weiteren Betrachtung vorgemerkt. Findet eine Strategie nach ihren Kriterien mehr als eine Gewinnerhypothese, so werden alle gefundenen Hypothesen als Gewinner markiert.

LOCKSET Der *LockSet*-Algorithmus in seiner ursprünglichen Form dient dem Auffinden von Wettlaufsituationen um Speicherzugriffe [SBN+97]. Wie bereits in Abschnitt 5.2.3 deutlich wurde, lässt sich die Idee so erweitern, dass damit Hypothesen abgeleitet werden können: Nach einer erfolgreichen Ausführung des zu untersuchenden Programms verbleiben in den internen Datenstrukturen des *LockSet*-Algorithmus die beobachteten Sperren-Mengen, die während aller Zugriffe eingesetzt wurden. Diese können als Sperren-Menge zur Absicherung der jeweiligen Speicherbereiche betrachtet werden. Auf den hiesigen Ansatz übertragen bedeutet es, dass die Hypothese mit einem relativen Support von 100 % und den meisten Sperren als Gewinner auszuwählen ist.

Beginnend bei dem Knoten „keine Sperren“ traversiert die Strategie den Graphen: Ist der aktuelle Knoten besser als der aktuelle Gewinner, wird der aktuelle Knoten der neue Gewinner und alle Nachfolgeknoten werden zur Betrachtung vorgemerkt. Ein Knoten ist besser, wenn der relative Support weiterhin 100 % beträgt, er aber mehr Sperren repräsentiert. Daher betrachtet die *LockSet*-Strategie in Abbildung 5.3 die Knoten v_0, a, b, c, ac und ab. Als Gewinner wird die Hypothese a ausgewählt.

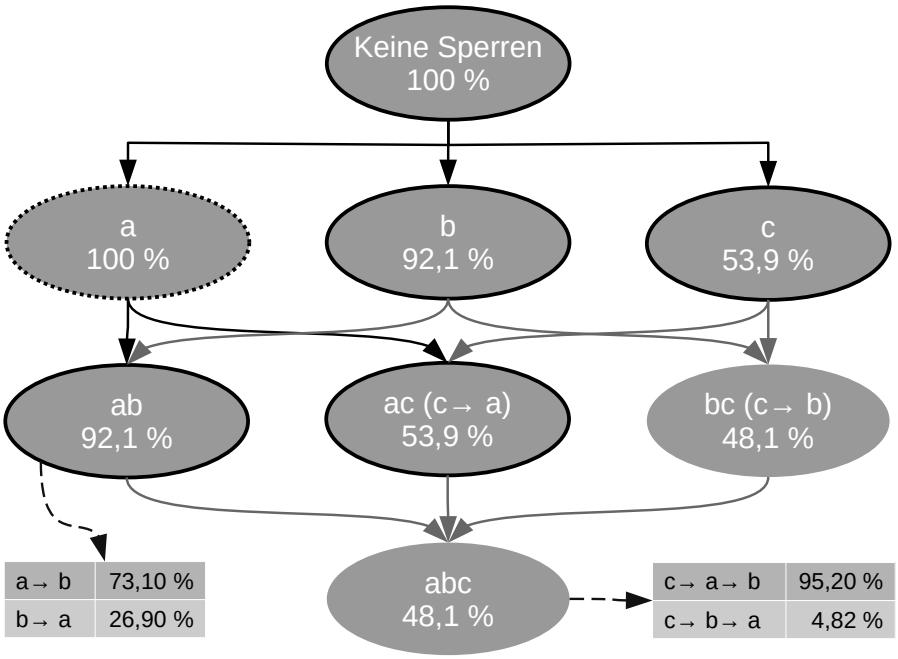


Abbildung 5.3: Visualisierung der Graph-Traversierung gemäß der Auswahlstrategien *Top Down*, *Sharpen* und *LockSet*. Den Ausgangspunkt für die Traversierung bildet der Graph in Abbildung 5.2. In Schwarz hervorgehoben wurden die Kanten und Knoten, die in Folge der Traversierung besucht werden. Gestrichelt hervorgehoben ist derjenige Knoten, welcher die Gewinnerhypothese repräsentiert.

TOP DOWN Die *Top Down*-Strategie wählt die Hypothese als Gewinner aus, die den größten relativen Support aufweist und gleichzeitig oberhalb eines Schwellenwerts liegt ($s_r \geq t_{ac}$). Eine Nebenbedingungen ist hier, dass es gleichzeitig nicht mehr als n_{nolock} % echte Zugriffe ohne Sperren gibt. Gibt es mehr als n_{nolock} % Zugriffe ohne jegliche Sperre, wird davon ausgegangen, dass keinerlei Sperren erforderlich sind. Treffen beide Bedingungen nicht zu, so gibt es keine Gewinnerhypothese. Nach dieser Strategie wird der Graph traversiert, solange der aktuell betrachtete Knoten besser ist. Hierzu muss der relative Support natürlich oberhalb des Schwellwerts liegen ($s_r \geq t_{ac}$) und er darf nicht schlechter sein als der vorherige beste Knoten. Bei der Behandlung des Knotens v_0 wird der Parameter n_{nolock} berücksichtigt. Da es in dem Beispiel nicht genügend Zugriffe ohne Sperre gibt, werden die Nachfolger von v_0 betrachtet. In Abbildung 5.3 ist das Vorgehen der Strategie für $t_{ac} = 0,95$ und $n_{nolock} = 0,05$ abgebildet. Es werden die Knoten v_0, a, b, c, ac und ab besucht, da der Knoten a zu erst begutachtet wird. Würde stattdessen nach v_0 der Knoten b besucht werden, so würde zusätzlich der Knoten bc betrachtet, da der Knoten b zwischenzeitlich als bester Knoten ausgewählt würde. Als Gewinner wird die Hypothese a ausgewählt.

BOTTOM UP Die *Bottom Up*-Strategie wählt die erste Hypothese als Gewinner aus, die noch oberhalb des Schwellwerts liegt ($s_r \geq t_{ac}$) und die meisten Sperren beinhaltet. Der Graph hingegen wird vom Ursprungskno-

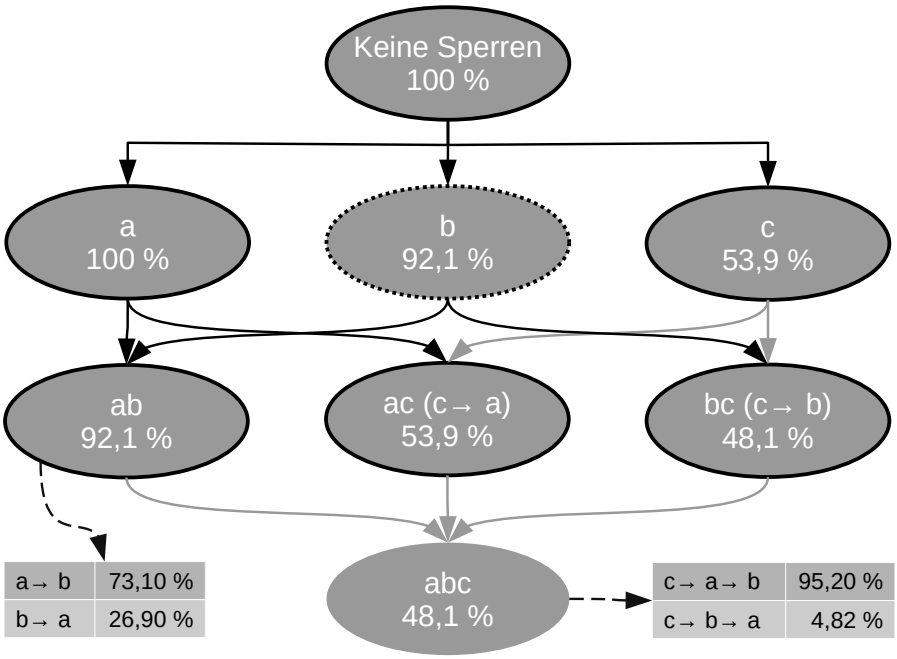


Abbildung 5.4: Visualisierung der Graph-Traversierung gemäß der Auswahlstrategie *Bottom Up*. Den Ausgangspunkt für die Traversierung bildet der Graph in Abbildung 5.2. In Schwarz hervorgehoben wurden die Kanten und Knoten, die in Folge der Traversierung besucht werden. Gestrichelt hervorgehoben ist derjenige Knoten, welcher die Gewinnerhypothese repräsentiert.

ten „keine Sperren“ solange durchlaufen bis sich kein Knoten finden lässt, dessen zugehöriger relativer Support oberhalb des Schwellwerts t_{ac} liegt. Im vorliegenden Beispiel werden die Knoten v_0, a, b, c, ac, ab und bc durchlaufen. Die Abbildung 5.4 zeigt den Durchlauf für $t_{ac} = 0,90$. Da die Hypothese $a \rightarrow b$ lediglich auf einen relativen Support von $0,921 * 0,731 = 0,6733$ kommt, wird sie nicht ausgewählt. Die Gewinnerhypothese lautet b. Wählt man jedoch $t_{ac} = 0,95$, so gewänne die Hypothese a.

SHARPEN Die *Sharpen*-Strategie nimmt beginnend bei einer leeren Sperren-Menge solange weitere Sperren zu der Gewinnerhypothese hinzu, solange die Verschlechterung des relativen Supports unterhalb eines Schwellenwerts bleibt: $1 - \frac{s_r(neu)}{s_r(alt)} \leq t_s$. Bei Hypothesen mit der gleichen Anzahl an Sperren wird die Hypothese mit dem größten relativen Support ausgewählt. Dieses Vorgehen ist in Abbildung 5.3 für $t_s = 0,1$ dargestellt. Beginnend beim Ursprungsknoten als erste potentielle Gewinnerhypothese werden zunächst die Knoten v_0, a, b, c, ac und ab besucht. Obwohl die Verschlechterung des relativen Supports deutlich oberhalb des Schwellenwertes liegt, wird der Knoten c dennoch besucht. Da sein Vorgängerknoten als bis dato bester Knoten ausgewählt wurde, wurden automatisch dessen Nachfolger vorgemerkt. Ähnlich verhält es sich für die Knoten ac und ab. Als letztendliche Gewinnerhypothese wird erneut die Hypothese a bestimmt.

6 LOCKDOC

Inhalt

In Kapitel 5 wurde die Grundidee hinter LockDoc in einem fiktiven Szenario erläutert. In diesem Kapitel erfolgt nun die Einordnung ebenjener Idee in einen konkreten Arbeitsablauf von der Aufzeichnung der Daten innerhalb des Betriebssystemkerns, über die Nachverarbeitung, das Ableiten der Sperren-Regeln bis hin zur Untersuchung der Dokumentation sowie des Programmcodes.

In der Abbildung 6.1 ist der eben skizzierte Ablauf visualisiert [LSBS19]:

❶ In der Phase *Monitoring/Tracing* wird das zu untersuchende Betriebssystem in einer virtuellen Maschine ausgeführt. Während der Ausführung einer bestimmten Arbeitslast werden folgende Informationen aufgezeichnet: Operationen (Belegen und Freigeben) auf den Sperren, Allokation und Freigabe von Speicherbereichen sowie den Zugriffen auf diese. Die Speicherbereiche gehören zu Instanzen der beobachteten Datenstrukturen. Die Ausgabe ist eine sequentielle Aufzeichnung aller Ereignisse, die während der Ausführung des Betriebssystems auftraten. Im Rahmen der Nachverarbeitung wird die Aufzeichnung mit weiteren Informationen angereichert und in eine relationale Datenbank überführt. Anschließend ist bereits bekannt, welche Elemente einer Datenstruktur zugegriffen wurden und welche Sperren dabei gehalten wurden. In Abschnitt 6.1 sowie Abschnitt 6.2 wird Phase 1 noch einmal detaillierter beleuchtet.

❷ Die Phase *Locking-Rule Derivation* entnimmt der Datenbank für jedes Element der untersuchten Datenstrukturen die bei den Zugriffen beobachteten Sperren-Mengen. Daraus wird in dem Werkzeug *Locking-Rule Derivator* eine Liste an Hypothesen für jedes Element generiert, wie dies bereits in Tabelle 5.3 zu sehen ist. Für jedes Paar aus dem Element einer Datenstruktur und des Zugriffstyps wird eine Gewinnerhypothese ausgewählt und in der Menge *Generated Locking Rules* abgelegt. Weiterführende Informationen zu diesem Schritt fanden sich bereits in Abschnitt 5.2.3.

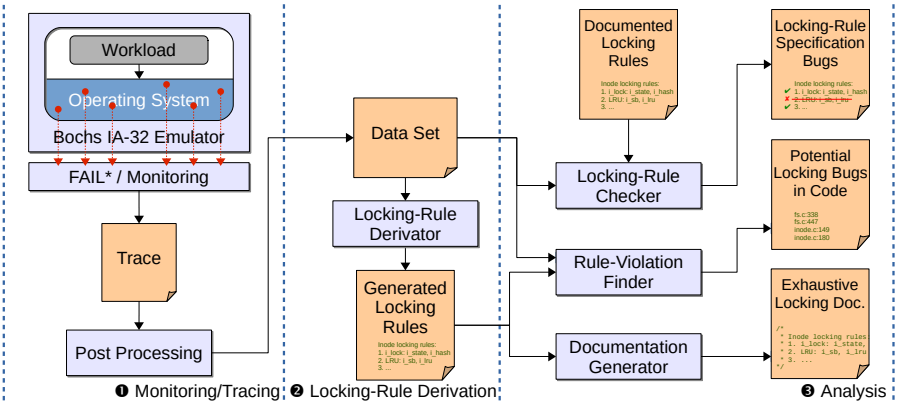


Abbildung 6.1: Übersicht über den Arbeitsablauf von LockDoc: Basierend auf den Aufzeichnungen der Speicherzugriffe und Sperren-Operationen aus dem Betriebssystemkern (Phase ❶) wird die wahrscheinlichste Sperren-Regel für eine definierte Menge an Datenstrukturen abgeleitet (❷). Mit diesen Informationen sucht LockDoc nach Fehlern in der Dokumentation, nach Programmierfehlern und generiert neue Sperren-Dokumentation (❸). (Aus einem Papier von Lochmann et al. entnommen [LSBS19])

❸ In der letzten Phase werden die Ergebnisse aus Phase ❶ und ❷ zusammengeführt. Zum einen wird in dem Schritt *Locking-Rule Checker* überprüft, inwieweit die existierende Sperren-Dokumentation zu dem Quellcode passt. Zum anderen wird aus der Menge der generierten Sperren-Regeln (*Generated Locking Rules*) eine aktualisierte Sperren-Dokumentation für die Entwickler erzeugt (*Documentation Generator*). Abschließend können mittels der erzeugten Sperren-Regeln und dem Datensatz Verletzungen der Sperren-Regeln gefunden und im Programmcode lokalisiert werden. Abschnitt 6.4 gibt einen tieferen Einblick in die drei Ergebnisse von LockDoc.

6.1 AUFZEICHNUNG

Das grundlegende Vorgehen des aufzeichnungsbasierten Ansatzes von LockDoc, wurde bereits in Unterabschnitt 5.1.1 erläutert. In diesem Abschnitt liegt der Fokus auf den technischen Aspekten während der Ausführung einer Arbeitslast. Das zu untersuchende Betriebssystem wird mittels des Fehlerinjektionswerkzeugs Fail* [SHD+15] (v1.0.1) in einer virtuellen Maschine ausgeführt [LSBS19]. Zum Einsatz kommt hierbei der x86-Emulator Bochs [Law96] (v2.4.6), welcher eine Ein-Kern-Maschine emuliert [LSBS19]. Das Werkzeug Fail* unterstützt bisher nur die Architektur i386 [SHD+15]. Die Ausführung des Gastsystems läuft hierbei vollkommen automatisiert ab [SHD+15]. Über ein sogenanntes Experiment wird in Fail* die Ausführung des Gastsystems gesteuert [SHD+15]. Darüber können Speicherzugriffe wie auch Zugriffe auf I/O-Ports überwacht werden [SHD+15]. Letztere dient als Kommunikationskanal für das Gastsystem. Im Programmcode des Ziel-

Ereignis	Gastsystem							Fail*-Experiment					
	Ereignistyp	Adresse	Größe	Code-Position	Datenstruktur	Sperren-Typ	Kontext	Zeistempel	Aufrufhierarchie	Ereignistyp	Adresse	Größe	Kontext
Belegen/Freigabe einer Sperre	x	x		x		x	x	x					
Allokation/Freigabe einer Datenstruktur	x	x	x		x		x	x					
Speicherzugriff (r/w)								x	x	x	x	x	x

Tabelle 6.1: Ein Überlick über die wesentlichen von LockDoc zu jedem Ereignis erhobenen Informationen. Eine Zeile listet die zu dem jeweiligen Ereignis ermittelten Informationen auf. Diese sind nach dem Ort der Aufzeichnung gruppiert: im Gastbetriebssystem selbst oder außerhalb im Fail*-Experiment.

Betriebssystems werden dafür die Sperren-Operationen[1] sowie die Allokation und Freigabe der beobachteten Datenstrukturen instrumentiert. Führt das Betriebssystem nun eine instrumentierte Stelle aus, werden die nötigen Informationen erhoben und in einen dafür vorgesehenen Speicherbereich geschrieben. Tabelle 6.1 gibt einen Überblick über die zu jedem Ereignis aufgezeichneten Informationen. Über eine Ausgabe auf dem I/O-Port 0xe9 signalisiert das Gastsystem nun, dass ein Ereignis eingetreten ist. Das Fail*-Experiment fängt dies ab [LSBS19]. Da die Ausführung des Gastsystems nun unterbrochen ist und die Kontrolle an das Fail*-Experiment übergeben wurde, werden hier je nach Ereignis zusätzliche Informationen erhoben – vgl. Tabelle 6.1. Hierzu zählt z. B. der Kontext eines Ereignisses, wie z. B. der Prozess oder aber die erste oder zwei Ebene der Unterbrechungsbehandlung. Dies wird im Rahmen der Nachverarbeitung in die Datenbank übertragen. Die Daten werden anschließend, mit einem Zeitstempel versehen, in eine CSV-Datei geschrieben [LSBS19]. Abbildung 6.2 zeigt einen Auszug aus einer solchen CSV-Datei für eine Aufzeichnung des Linux-Kerns. Anschließend übergibt das Experiment die Kontrolle wieder an das Gastsystem. Zusätzlich wird Fail* bei Allokations-Ereignissen angewiesen den allozierten Speicher hinsichtlich Zugriffen zu beobachten [LSBS19]. Analog dazu wird der Speicherbereich nicht mehr beobachtet, wenn die Freigabe signalisiert wurde [LSBS19]. Somit werden die Speicherzugriffe von außen über das Fail*-Experiment ebenfalls aufgezeichnet und mit Zeitstempeln verse-

1 Genaue Informationen zu der Instrumentierung liefern die Abschnitt 7.2.2 bzw. Abschnitt 7.3.1.

```
1 255960950#l#1#0xc20102a8#NULL#NULL#raw_spinlock_t#&logbuf_lock#kernel/printk/
      printk.c#2499#console_unlock#NULL#0x1#0#NULL#0#0
2 1255961920#l#3#0xc20102a8#NULL#NULL#raw_spinlock_t#&logbuf_lock#kernel/printk
      /printk.c#2501#console_unlock#NULL#0x1#0#NULL#0#0
3 2070728067#a#NULL#0xf5424400#896#NULL#super_block#NULL#NULL#NULL#NULL#NULL
      #0#0#NULL#0#0
4 2070728074#w#NULL#0xf542446c#4#0xf5424400#NULL#NULL#NULL#NULL#NULL#0xc137958e
      #NULL#NULL#0xc137cd19,0xc138156a,[...]
```

Abbildung 6.2: Ein Auszug aus einer Aufzeichnung des Linux-Kerns. Hier dargestellt ist das Belegen (`#l#1#`) sowie Freigeben (`#l#3#`) einer Schreiber-Sperre. Dazu gehört u.a. die Speicheradresse der Sperre wie auch die Stelle im Quellcode, an der die Operationen ausgeführt wurden. Außerdem wird die Allokation (`#a#`) und das Schreiben (`#w#`) der Größe 4 dargestellt. Zu einem Speicherzugriff wird u.a. die Aufrufhierarchie aufgezeichnet. Der Wert `NULL` dient als Platzhalter, damit jede Zeile die gleiche Anzahl an Spalten aufweist.

hen [LSBS19]. Das Beispiel aus Abbildung 4.1 wird somit auch abgedeckt. Das Fail*-Experiment umfasst 744 Zeilen[2] an *C++*-Code.

6.2 NACHVERARBEITUNG

Die sequentielle Abfolge von Ereignisse aus Phase ❶ wird nun in einem mehrstufigen Prozess in eine relationale Datenbank überführt [LSBS19]. Die wesentlichen Bestandteile des zugehörigen Datenbankschemas sind in Abbildung 6.3 abgebildet. Die zentrale Tabelle *accesses* beinhaltet alle Speicherzugriffe, die während der Ausführung der virtuellen Maschine aufgezeichnet wurden [LSBS19]. Mittels der *DWARF*[3]-Informationen aus dem Kern-Abbild sowie den aufgezeichneten Allokationen und Freigaben lassen sich die Zugriffe einzelnen Allokationen (Tablle *allocations*) zuordnen [LSBS19]. Eine Allokation umfasst dabei die Lebensdauer eines Speicherbereichs [LSBS19]. Eine lässt sich über die Ereignisse *Allokation* und *Freigabe* einer Datenstruktur zuordnen [LSBS19]. Schlussendlich dienen die *DWARF*-Informationen ebenfalls zur Rekonstruktion (Tabelle *tuple_layout*) der beobachteten Datenstrukturen, so dass einzelne Speicherzugriffe letztlich einzelnen Elementen einer Datenstruktur zugeordnet werden können [LSBS19]. Zusätzlich wird für jeden Speicherzugriff die genaue Aufrufhierarchie vom Eintritt in den Betriebssystemkern bis zu dem betreffenden Zugriff aufgezeichnet (Tabelle *stack_traces*) [LSBS19]. Dies wird in dem Absatz Unterabschnitt 6.4.3 genutzt, um die exakte Position zu bestimmen, an der ein falsch synchronisierter Speicherzugriff geschehen ist [LSBS19].

Erfolgte ein Speicherzugriff innerhalb einer Transaktion, so wird er dieser zugeordnet (Tabelle *txns*) und deren Dauer vermerkt [LSBS19]. Da jeder Transaktion eine feste Menge an Sperren zugeordnet sind, ist hiermit für

2 Anzahl der effektiven Zeilen Code, ohne Leerzeilen und Kommentare. Ermittelt mittels *cloc* (https://github.com/AlDanial/cloc).

3 http://dwarfstd.org/

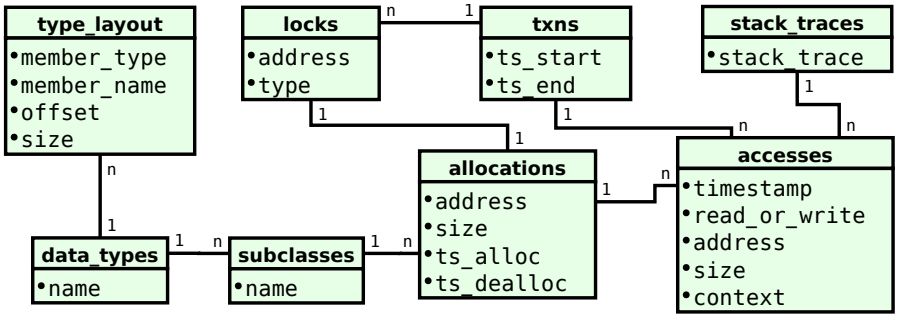


Abbildung 6.3: Das Datenbankschema von LockDoc bildet die aufgezeichneten Daten strukturiert über mehrere Tabellen verteilt ab. Die zentrale Tabelle bildet hierbei die Tabelle *accesses*, welche die einzelnen Speicherzugriffe darstellt. Über diese Tabelle kann jeder Speicherzugriff mit Kontextinformationen versehen werden. (Aus einem Papier von Lochmann et al. entnommen [LSBS19])

jeden Speicherzugriff eindeutig die Sperren-Menge festgelegt [LSBS19]. Anhand des Beispiels in Abschnitt 5.2.2 wurde bereits deutlich, dass sich mit jeder weiteren geholten Sperre ein Transaktionsstapel bildet. Dieser wird mit der Freigabe einer Sperre sukzessive wieder abgebaut. Werden allerdings Sperren eingesetzt, die das Schlafen (Prozesswechsel) innerhalb eines kritischen Abschnitts erlauben, so werden auf einem Stapel Transaktionen aus verschiedenen Kontexten abgelegt. Wurden nun Speicherzugriffe innerhalb dieser Transaktionen aufgezeichnet, führt dies zu synthetischen Sperren-Kombinationen, die es so im Programmablauf nicht gibt. Da sie nun aber mindestens eine Beobachtung haben, müssen sie bei der Auswertung berücksichtigt werden und erhöhen das Grundrauschen. Alternativ kann für jeden Ausführungskontext ein separater Transaktionsstapel verwendet werden. Hierbei wird vermieden, dass es künstliche Sperren-Kombinationen gibt, so dass es die Auswertung verkürzt. Ist es bei der Entwicklung beabsichtigt, eine Sperre in Kontext 1 zu holen und in Kontext 2 wieder freizugeben, so muss die betroffene Transaktion von dem betreffenden Stapel „gestohlen“ werden. Andernfalls könnte der Transaktionsstapel nie vollständig abgebaut werden. Dies kann beispielsweise beim Kontextwechsel auftreten, wenn die dafür zuständige Datenstruktur für die Dauer des Wechsel geschützt werden muss. LockDoc unterstützt letztlich beide Vorgehensweisen. Sie werden in Unterabschnitt 7.2.3 bzw. Unterabschnitt 7.3.2 evaluiert.

Für jede Sperre (Tabelle *locks*) wird außerdem vermerkt, ob es sich um eine globale Sperre handelt oder ob sie in eine Datenstruktur eingebettet ist [LSBS19]. Ist letzteres der Fall kann ebenfalls bestimmt werden, ob die Sperre in die gleiche Datenstruktur eingebettet ist, die gerade zugegriffen wird [LSBS19].

Neben den bereits geschilderten Verarbeitungsschritten gibt es noch drei spezielle Schritte bei der Nachverbarbeitung, die dem Umstand geschuldet sind, dass hier reale Software-Projekte untersucht werden:

1. Wenngleich Programmiersprachen wie *C* keine Objektorientierung und damit verbunden keine Unterklassen bereitstellen, so lassen sich

In Kapitel 7 werden Angaben zu der Menge der gefilterten Funktionen und Elemente gemacht.

diese Mechanismen doch nachbilden [LSBS19]. Im Linux-Kern beispielsweise wird dies bei der zentralen Datenstruktur `struct inode` zur Verwaltung von Dateisystemobjekten umgesetzt [LSBS19]. Hier existiert das Element `i_private`, das für den privaten Gebrauch durch das jeweilige Dateisystem, das sich hinter einer Instanz der Datenstruktur verbirgt, gedacht ist [LSBS19]. Daher gibt es keine einheitliche Sperren-Regel für dieses Element [LSBS19]. Vielmehr wird durch den Dateisystemtreiber festgelegt, ob und wie Zugriff auf dieses Element geschützt werden [LSBS19]. Ähnlich verhält es sich mit anderen Elementen der Datenstruktur `struct inode`: Das Dateisystem *procfs* schützt Teile der Datenstruktur gar nicht mit Sperren, da es nur eine Teilmenge der Dateisystemoperationen implementiert und somit keine Sperren erforderlich sind [LSBS19]. Aus diesen Gründen erlaubt LockDoc die Differenzierung der Datentypen in Unterklassen (Tabelle *subclasses* in Abbildung 6.3) [LSBS19]. Somit werden Sperren-Regeln für jede Unterklasse abgeleitet, wie z. B. *procfs* oder *ext4* [LSBS19].

2. In einer idealen Welt wird jeder Zugriff nach dem gewählten Sperren-Regeln abgesichert [LSBS19]. Da ein Objekt während der Erzeugung bzw. Initialisierung sowie der Zerstörung dem restlichen System (noch) nicht zur Verfügung steht, wird hier meist von dem Einsatz von Sperren abgesehen [LSBS19]. Solche Beobachtungen beeinflussen jedoch entweder die abgeleiteten Sperren-Regeln oder werden später als potentielle Fehler aufgezeigt [LSBS19]. Daher wird für jede Datenstruktur eine Liste mit diesen Funktionen geführt und Zugriffe im Kontext dieser Funktionen werden gefiltert [LSBS19].

3. Abschließend werden Zugriffe auf Elemente gefiltert, die für die Analyse als nicht relevant anzusehen sind [LSBS19]. Dazu zählen z. B. die in die Datenstruktur eingebetteten Sperren selbst, aber auch verschachtelte Datenstrukturen [LSBS19]. Ferner werden Zugriffe auf Elemente, die bereits atomar zugegriffen werden können, gefiltert [LSBS19]. Hierzu zählt z. B. der Datentyp `atomic_t` [Lov10] im Linux-Kern [LSBS19]. Natürlich können Elemente solchen Typs Bestandteil einer größeren kritischen Abschnitts sein, der wiederum Sperren erfordert [LSBS19]. Da bis hierher nur Sperren-Regeln für jedes Element einzeln abgeleitet werden, werden Zugriffe auf solche Elemente aber gefiltert (vgl. Abschnitt 5.2.3) [LSBS19].

Die für die Punkte 2. und 3. erforderlichen Listen müssen anhand des Programmcodes manuell erstellt und im Zuge der Nachverarbeitung in die Datenbank übernommen werden. Die zugehörigen Tabellen sind wegen der besseren Übersicht in Abbildung 6.3 nicht abgebildet.

Die gesamte Nachverarbeitung wird durch ein *Shell*-Skript koordiniert. Dabei erfolgt das Auswerten der Ereignisse und Anreichern mit *DWARF*-Informationen in 7.973 Zeilen *C*- und *C++*-Code [LSBS19]. Die hieraus resultierenden CSV-Tabellen werden in eine *PostgreSQL*[4]-Datenbank übernom-

4 https://www.postgresql.org/

men. Die beschriebene Filterung der Daten wird über die *SQL*-Anfragen realisiert.

6.3 ABLEITEN DER SPERREN-REGELN

Auf die Nachverarbeitung folgt – ebenfalls noch in Phase ❷ – das Ableiten der Sperren-Regeln [LSBS19]. Hierzu wird eine Liste aller Speicherzugriffe gruppiert nach Zugriffstyp, Element und Datenstruktur sowie die jeweils gehaltenen Sperren aus der Datenbank abgefragt [LSBS19]. Die zuvor erwähnten Filter werden hierbei angewendet [LSBS19].

Das in Abbildung 6.1 mit *locking-rule derivator* bezeichnete Werkzeug leitet aus diesen Daten die Sperren-Regeln ab [LSBS19]. Es ist in *C++* realisiert und um fasst 2.280 Zeilen *C++*-Code[5]. Die Ausführung kann über verschiedene Parameter wie z. B. die Wahl von t_{ac}, die Methode zur Bestimmung der Gewinnerhypothese oder aber verschiedene Ausgabemodi kontrolliert werden [LSBS19].

Das Werkzeug generiert für jedes Tupel aus Zugriffstyp, Element und Datenstruktur die Liste aller möglichen Hypothesen und ermittelt für jede Hypothese, wie in Abschnitt 5.2.3 erläutert, die Metriken s_a und s_r [LSBS19]. Wie bereits anhand der Abbildung 1.1 bzw. 3.1 zu sehen war, enthalten komplexe Softwaresysteme wie ein Betriebssystemkern eine Vielzahl an Sperren. Anstatt die Metriken für alle theoretisch möglichen Teilmengen zu bestimmen, iteriert das Werkzeug *locking-rule derivator* nur über alle beobachteten Sperren-Regeln (hier Sperren-Kombinationen) und generiert daraus alle möglichen Teilmengen und Abfolgen [LSBS19]. So ist sichergestellt, dass alle sinnvollen Hypothesen ($s_a > 0$) berücksichtigt werden [LSBS19]. Das dazu erforderliche Vorgehen ist in dem Algorithmus 6.1 dargestellt. Zunächst wird aus der Sperren-Menge *sm* zu jeder beobachteten Sperren-Kombination *lk* (Zeile 3) die Potenzmenge M erzeugt. Diese beinhaltet alle möglichen Teilmengen der Länge 0, 1, 2, 3,... . Eine Sperren-Kombination kann beispielsweise $d \rightarrow f \rightarrow e$ sein. Die zugehörige Sperren-Menge lautet $\{d, e, f\}$. Im Anschluss wird für jede generierte Teilmenge $m \in M$ überprüft, ob sie eine Teilmenge der Sperren-Menge $lk2.sm$ ist (Zeilen 9 und 10). Ist dies der Fall wird eine Hypothese h angelegt. Weiter werden die gezählten Beobachtungen der gerade untersuchten Sperren-Kombination $lk2$ der Hypothese h zugeschlagen. Aus der Sperren-Reihenfolge $lk2.sr$ wird mittels der Teilmenge m die Sperren-Reihenfolge sr_ext extrahiert (Zeile 12 ff.). Für die Teilmenge $\{d, e\}$ würde aus dem oben genannten Beispiel die Sperren-Reihenfolge $d \rightarrow e$ extrahiert werden. Ist diese Sperren-Reihenfolge sr_ext bereits unter der Hypothese vermerkt, wird die Anzahl der Beobachtungen $lk2.b$ der Sperren-Reihenfolge zugeschlagen (Zeile 13 ff.). Andernfalls wird die Anzahl der Beobachtungen $lk2.b$ unter der Sperren-Reihenfolge sr_ext in der Map $h.k$ abgelegt. Auf diese Weise fließen auch Beobachtungen ein, die über mehr Sperren verfügen als die gerade zu bewertende Hypothese eigentlich

5 Anzahl der effektiven Zeilen Code, ohne Leerzeilen und Kommentare. Ermittelt mittels *cloc* (https://github.com/AlDanial/cloc).

Algorithmus 6.1 : Aufstellen aller relevanten Hypothesen ($s_a > 0$) für ein Tupel aus Datenstruktur, Element und Zugriffstyp

Datenstrukturen :
Sperren-Kombination: Anzahl Beobachtungen b, die Sperren-Menge sm, die Sperren-Reihenfolge sr.
Hypothese: Anzahl Beobachtungen b, die Map k ordnet einer Sperren-Reihenfolge ihren Beobachtungen zu.
Eingabe : Sei B die Liste aller beobachteten Sperren-Kombinationen.
Ausgabe : Sei H eine Map und ordne eine Sperren-Menge einer Hypothese zu.

```
1  Beginn
2      für jedes lk aus B tue
3          Bestimme die Potenzmenge M von lk.sm;
4          für jedes m ∈ M tue
5              wenn m in H existiert dann
6                  Gehe weiter zu der nächsten Teilmenge;
7              Erstelle neue Hypothese h;
8              H[m] := h;
9              für jedes lk2 aus B tue
10                 wenn m ⊆ lk2.sm dann
11                     h.b := h.b + lk2.b;
12                     sr_ext := extrahierte Sperren-Reihenfolge für m
                        aus lk2.r;
13                     wenn sr_ext in h.k existiert dann
                           /* Aufaddieren der Beobachtungen      */
14                         h.k[sr_ext] := h.k[sr_ext] + lk2.b;
15                     sonst
                           /* Einfügen der neuen
                              Sperren-Reihenfolge                */
16                         h.k[sr_ext] := lk2.b;
```

enthalten [LSBS19]. Sagt eine Hypothese beispielsweise aus, dass die Sperre a vor der Sperre b zu holen ist – $a \longrightarrow b$ –, so werden Beobachtungen, die beispielsweise zusätzlich die Sperre c beinhalten, wie $c \longrightarrow a \longrightarrow b$ oder $a \longrightarrow c \longrightarrow b$, ebenfalls dieser Hypothese zugerechnet. Beide Beobachtungen erfüllen die Hypothese „a vor b“ [LSBS19].

Die Ausgabe H beinhaltet schließlich für ein Tupel aus Zugriffstyp, Element und Datenstruktur eine Map, die jeder Sperren-Menge ihren absoluten Support s_a sowie alle beobachteten Sperren-Reihenfolgen inkl. des zugehörigen absoluten Supports zuordnet. Daraus wird für jedes Tupel eine Gewinnerhypothese ermittelt [LSBS19]. Diese enthält den Namen der Sperre sowie die Information, ob es sich bei der Sperre um eine globale Variable handelt oder sie in eine Datenstruktur eingebettet ist. Ein ähnliche Differenzierung schlugen bereits Breuer et al. im Ausblick ihrer Arbeit vor [BV04]. Ferner kann noch unterschieden werden, ob die Sperre in die gerade zugegriffene Datenstruktur oder irgendeine andere eingebettet ist. Die Regel

gibt außerdem an, welche Teil-Sperre, Leser- oder Schreiber-Sperre, zu verwenden ist. Eine Hypothese kann wie folgt aussehen: `inode_hash_lock[w] -> EMBSAME(inode.i_rwsem[w])-> EMBOTHER(inode.i_rwsem[r])`. Hier ist zunächst die Schreiber-Sperre der globalen Sperre `inode_hash_lock` erforderlich. Anschließend folgt ebenfalls die Schreiber-Sperre `EMBSAME(inode.i_rwsem[w])`, die in die gerade verwendete Datenstruktur eingebettet ist. Abschließend ist eine Leser-Sperre zu holen, die in irgendeiner anderen Instanz der Datenstruktur `struct inode` integriert ist.

6.4 ANALYSE

In Phase ❸ werden die abgeleiteten Sperren-Regeln zur weiteren Analyse des Programmcodes sowie der Sperren-Dokumentation eingesetzt. [LSBS19] Hierzu werden außerdem die strukturierten Informationen aus der Datenbank herangezogen [LSBS19]. Die Analyse lässt sich in die folgenden drei Themenbereiche gliedern, die in Kapitel 7 auf die zu untersuchenden Betriebssystemkerne angewendet werden:

6.4.1 *Überprüfung der Sperren-Regeln*

In diesem Schritt wird die existierenden Sperren-Dokumentation überprüft, ob sie noch zu den Beobachtungen passt (vgl. *Locking-Rule Checker* in Abbildung 6.1) [LSBS19]. Das Vorgehen ist in Algorithmus 6.2 dargestellt. Es wird die Sperren-Dokumentation der beobachteten Datenstrukturen herangezogen [LSBS19]. Zunächst muss die Dokumentation zunächst aus dem betreffenden Programmcode manuell extrahiert und in ein für LockDoc verständliches Format übersetzt werden [LSBS19]. Wichtig ist, dass hier nur die Regeln berücksichtigt werden, die an zentralen Stellen, wie die Definition der Datenstruktur oder zu Beginn von wichtigen *C*-Dateien, zu finden sind. Dokumentation, die verstreut über den Quelltext ist, wird ignoriert, da hier ein Entwickler auch nicht zu erst suchen würde. Für jede extrahierte, dokumentierte Sperren-Regel wird in der Liste der Hypothesen der relative Support s_r nachgeschaut [LSBS19]. Anschließend wird anhand des relativen Supports eine Kategorisierung vorgenommen: korrekt ($s_r = 1$), ambivalent ($0 < s_r < 1$) und falsch ($s_r = 0$) (Zeilen 6, 8 und 10) [LSBS19]. Die Kategorien ambivalent und falsch bedeuten, dass die Sperren-Regeln im Programmcode gar nicht oder unzureichend umgesetzt werden [LSBS19]. In Folge dessen sollte die Dokumentation durch die Entwickler aktualisiert oder durch die von LockDoc generierte Dokumentation ergänzt werden [LSBS19]. Findet sich für ein Tupel keine Hypothese, wird dies ebenfalls gezählt (Zeile 12). Umgesetzt wurde die Überprüfung der Sperren-Dokumentation in 129 Zeilen *Python*-Code.

Algorithmus 6.2 : Überprüfung der Sperren-Regeln. Der Algorithmus ermittelt den Anteil der korrekt ($s_r = 1$), ambivalent ($0 < s_r < 1$) und gar nicht ($s_r = 0$) im Programmcode umgesetzten Sperren-Regeln. Das hier dargestellte Vorgehen beschränkt sich der Einfachheit halber auf die Berechnung der Werte für eine Datenstruktur.

Datenstrukturen : Jede Map ordnet einem Tupel aus Datentyp, Element und Zugriffstyp einer Sperren-Regel bzw. Hypothese zu.
Eingabe : G ist eine Map der dokumentierten Sperren-Regeln für eine Datenstruktur,
H ist eine Map *aller* beobachteten Hypothesen für eine Datenstruktur.
Ausgabe : g ist die Anzahl gefundenener Tupel
a ist die Anzahl Tupel mit $s_r = 1$
b ist die Anzahl Tupel mit $s_r < 1$ && $s_r > 0$
c ist die Anzahl Tupel mit $s_r = 0$
ng ist die Anzahl nicht gefundener Tupel

```
1  Beginn
       /* Initialisiere alle Ausgabevariablen mit 0          */
2      für jedes Tupel t mit der Sperren-Regel r aus G tue
3          wenn t in H existiert dann
4              g := g + 1;
5              wenn s_r(r) == 1 dann
6                  a := a + 1;
7              sonst wenn s_r(r) < 1 && s_r(r) > 0 dann
8                  b := b + 1;
9              sonst
10                 c := c + 1;
11         sonst
12             ng := ng + 1;
```

6.4.2 *Generierung von Sperren-Regeln*

In diesem Schritt werden für alle nicht-ignorierten Elemente aller untersuchten Datenstrukturen gemäß der Methodik aus Abschnitt 5.2.3 die Sperren-Regeln bestimmt [LSBS19]. Die so gewonnenen Regeln können nun zum automatischen Erzeugen von Sperren-Dokumentation (vgl. *Documentation Generator* in Abbildung 6.1) und zum Auffinden von Fehlern (vgl. *Rule-Violation Finder* in Abbildung 6.1) genutzt werden [LSBS19]. Die Generierung von neuer Dokumentation ist in Form eines anderen Ausgabeformats in das Werkzeug *lockrng-rule derivator* integriert [LSBS19]. Ein Beispiel hierfür ist in Listing 6.1 zu sehen.

```
/*
 * inode locking rules:
 *
 * No locks needed for:
 *   w:i_op, w:i_link, w:i_ino, [...]
 *
 * ES(inode.i_rwsem[w]) protects:
 *   w:i_version, r:i_data.private_data
 *
 * ES(inode.i_rwsem[w]) -> rcu[r] protects:
 *   r:i_default_acl
 *
 * EO(super_block.s_umount[r]) protects:
 *   r:i_data.writeback_index
 *
 * inode_hash_lock(raw_spinlock_t[w]) protects:
 *   w:i_hash
 * [...]
*/
```

Listing 6.1: Ein exemplarischer Auszug aus der von LockDoc erzeugte Sperren-Dokumentation, wie sie zu Beginn der Datei `fs/inode.c` im Linux-Kern zu finden wäre. Die Abkürzungen *EO* („embedded other“) bedeutet eine Sperre, die in andere Datenstruktur eingebettet ist. *ES* („embedded same“) hingegen zeigt eine Sperre in die gerade zugegriffene Datenstruktur an. (Nach einer Vorlage von Lochmann et al. [LSBS19])

6.4.3 *Verstöße gegen Sperren-Regeln*

Im Gegensatz zum ersten Schritt wird nun angenommen, dass die erzeugten Sperren-Regeln korrekt sind [LSBS19]. Sie werden genutzt, um anhand aufgezeichneten Informationen Stellen im Programmcode zu identifizieren, die die abgeleitete Sperren-Regel nicht einhalten: die Gegenbeispiele [LSBS19]. Darunter sind alle Zugriffe zu verstehen, die weniger Sperren oder Sperren in der falschen Reihenfolge belegen als gemäß Sperren-Regel erforderlich wären. Sollten mehr Sperren involviert sein, als vorhergesagt wurde, wird dies nicht als Gegenbeispiel angesehen. Solche Fälle schränken zwar den möglichen Grad an Parallelität ein, sie sorgen jedoch nicht für Datenkorruption.

Ein Gegenbeispiel setzt sich aus der Aufrufhierarchie aus Datei, Zeile und Funktion, die zu diesem Speicherzugriff führte, sowie den eigentlich zu haltenden sowie den tatsächlich gehaltenen Sperren zusammen. Für jedes Tupel aus Zugriffstyp und Element werden für den Entwickler in Form einer *HTML*-Webseite[6], vgl. Abbildung 6.4, alle Gegenbeispiele eines Datentyps aufbereitet [LSBS19]. Hierbei wird bereits berücksichtigt, wenn ein Gegenbeispiel eine restriktivere Sperre nutzt als die Sperren-Regel vorschreibt. Dieser Fall kann bei einer Leser-Schreiber-Sperre auftreten: Die Regel sagt

6 Die Darstellung wurde mittels der Graph-Bibliotheken *Cytoscape.js* [FLH+15] und *Treant.js* [tre21] realisiert [MR18].

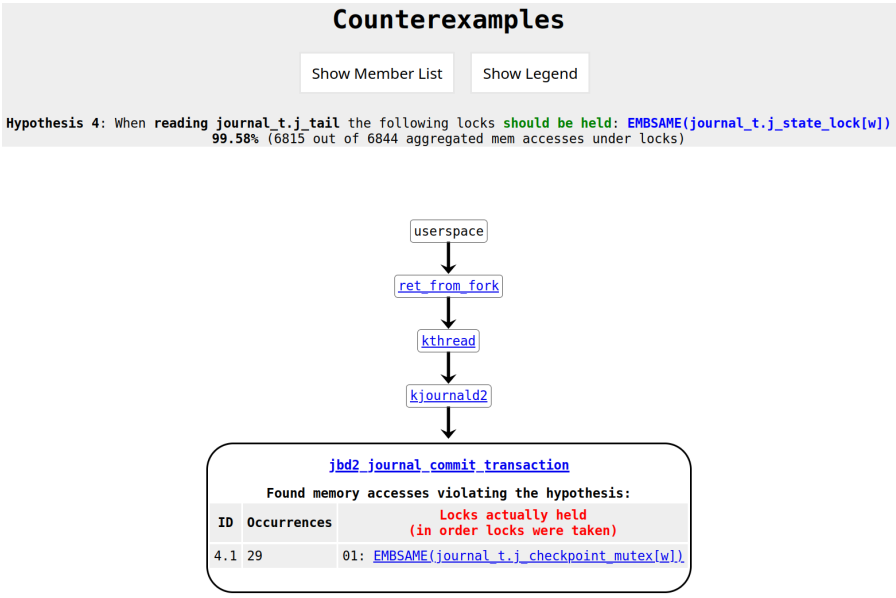


Abbildung 6.4: Exemplarische Visualisierung von Gegenbeispielen für das Element `j_tail` der Struktur `struct journal_s`. Zu sehen ist die Aufrufhierarchie, die zu den Speicherzugriffen führte. Das Blatt des Graphen steht für die zugreifende Funktion. Hier werden ebenfalls die verschiedenen, beobachteten Sperren-Kombinationen aufgelistet. Hinter jeder Funktion wie auch hinter jeder Sperre verbirgt sich ein Link, an die jeweilige Stelle im Quellcode. Die Gewinnerhypothese ist zu Beginn aufgeführt.

aus, dass die Leser-Sperre verwenden ist. In dem vermeintlichen Gegenbeispiel wird jedoch die Schreiber-Sperre eingesetzt. Solche Situationen werden nicht als Fehler bzw. Gegenbeispiele gezählt.

Realisiert wurde dieser Schritt mittels 603 Zeilen *Python*-Code sowie eines parametrisierbaren *Shell*-Skriptes, das eine passende *SQL*-Anfrage für die *PostgreSQL*-Datenbank generiert.

Wie der vorangegangenen Beschreibung zu entnehmen war, liegt der Fokus bei LockDoc auf den Sperren-Regeln. Damit lässt sich die existierende Dokumentation überprüfen, neue Dokumentation generieren sowie das aktuelle Programmverhalten auf Fehler untersuchen. Dies schließt auch das Auffinden von möglichen Wettlaufsituationen um Speicherzugriffe (*Data Races*) mit ein, da hier abweichend von der eigentlichen Sperren-Regel zu wenig Sperren gehalten werden. Wobei diese Fälle nicht explizit gekennzeichnet werden. Es ist aber nur ein Teil dessen, was mit LockDoc untersucht wird. Natürlich lässt sich anhand der Daten von LockDoc nicht mit absoluter Gewissheit sagen, ob jeweils wirklich eine Wettkampfbedingung vorliegt. Hier ist die Grenze zu anderen Arbeiten, wie z. B. *Razzer* [JKS+19] oder *Eraser* [SBN+97], zu ziehen. Sie zielen zwar nur auf das Auffinden bzw. Provozieren von Wettlaufsituationen ab. Allerdings zeigen sie damit auf, wo es definitiv zu einer Wettlaufsituation kommt. Sie bestimmen jedoch nicht, die Sperren-Mengen, die für eine korrekte Absicherung erforderlich wären.

7 FALLSTUDIEN

Inhalt

In Kapitel 5 wurde das grundlegende Vorgehen zur Auswertung der Daten vorgestellt. Dies wurde in Kapitel 6 in einen kompletten Arbeitsablauf integriert. Abschließend soll daher in diesem Kapitel die Vorstellung von zwei Fallstudien erfolgen: Für den in Kapitel 6 dargelegten Ablauf wurden die Betriebssystemkerne von Linux und FreeBSD angepasst und mittels LockDoc untersucht.

In Abschnitt 7.1 wird zunächst ein Plausibilitätstest für den LockDoc-Ansatz vorgestellt, der ebenfalls auf beide Betriebssystemkerne portiert wurde. Darauf folgt die Darstellung der Fallstudien für Linux und FreeBSD. Für beide Betriebssysteme untergliedert sich die Untersuchung in folgende Bereiche: Zu Beginn in Abschnitt 7.2.2 sowie Abschnitt 7.3.1 werden jeweils die erforderlichen Anpassungen an den jeweiligen Kern vorgestellt. Dies beinhaltet auch die Auflistung der untersuchen Daten- wie auch die unterstützten Sperren-Typen und schließt mit den Limitierungen für den jeweiligen Kern. Darauf folgen die eigentlichen Ergebnisse der Fallstudien (Abschnitt 7.2.4 bzw. Abschnitt 7.3.3) untergliedert nach den drei Kategorien, wie sie in Abschnitt 6.4 gezeigt wurden.

7.1 PLAUSIBILITÄTSPRÜFUNG

Zur Prüfung der Ergebnisse auf Plausibilität wird in diesem Abschnitt ein LockDoc-Test vorgestellt. Für diesen Zweck wird ein Test-Szenario mit einem Ringpuffer und entsprechender Synchronisation eingesetzt. Es soll geprüft werden, ob die erdachten Sperren-Regeln und vorhergesagten Zugriffsmuster auf den Ringpuffer von LockDoc korrekt ermittelt werden. Die Syn-

```
1 struct lockdoc_ring_buffer {
2         int next_in;
3         int next_out;
4         int size;
5         int data[201];
6 }
```

Listing 7.1: Darstellung der Ringpuffer-Datenstruktur für den LockDoc-Test. Der Puffer bietet Platz für 200 Elemente.

Element	Zugriffstyp	Sperren-Regel
data	r	producer_lock⟶rb_lock
	w	consumer_lock⟶ rb_lock
next_in	r	rb_lock
	w	producer_lock⟶rb_lock
next_out	r	rb_lock
	w	consumer_lock⟶ rb_lock
size	r	rb_lock

Tabelle 7.1: Die Sperren-Regeln für den LockDoc-Ringpuffer aufgeteilt nach Element und Zugriffstyp. Da die Größe des Puffers zum Initialisierungszeitpunkt feststeht, wird das Beschreiben der Variable `size` hier ignoriert.

chronisationsmuster sind hierbei bewusst so gewählt, um LockDoc zu überprüfen. Sie stellen eine sehr pessimistische Synchronisation der Zugriffe dar. Für einen effizienten und asynchronen Zugriff auf einen Ringpuffer wären weniger Sperren erforderlich. Die Definition des Ringpuffers ist in Listing 7.1 dargestellt. Er fasst 200 Elemente.[1] Zur Synchronisation des Zugriffs werden drei Sperren eingesetzt: Die Sperren `producer_lock` und `rb_lock` schützen den schreibenden Zugriff auf die Daten sowie das Beschreiben des Lese-Zeigers. Die Sperren `consumer_lock` und `rb_lock` schützen den lesenden Zugriff auf die Daten und das Modifizieren des Schreibe-Zeigers. Die Sperre `rb_lock` schützt hingegen das Lesen des Zustands (`next_in`, `next_out` und `size`). Daraus ergeben sich die in Tabelle 7.1 dargestellten Sperren-Regeln. Zur Überprüfung des LockDoc-Ansatzes wurden absichtlich unterschiedliche Regeln für das Lesen und Schreiben der Zustandsvariablen gewählt.

Der Quelltext für das Produzieren wie Konsumieren von Pufferelementen ist in Listing 7.2 sowie Listing 7.3 abgebildet. Alle vier Funktionen aus Listing 7.2 sowie Listing 7.3 werden jeweils 200-mal mit einer korrekten Synchronisation ausgeführt. Dabei werden die Sperren vor dem Aufruf einer jeden Funktion geholt und anschließend freigegeben.

Dies repräsentiert die Annahme aus Kapitel 5, dass die Zugriffe meistens korrekt ablaufen. Zusätzlich finden folgende, falsche Zugriffe statt. Dabei wird jedes Szenario genau einmal ausgeführt:

1 Aufgrund der gewählte Implementierung muss das Datenfeld ein Element mehr als die vorgesehene Größe fassen.

```c
int is_full(struct lockdoc_ring_buf *buf) {
     return (buf->next_in + 1) % buf->size == buf->next_out;
 }

int produce(struct lockdoc_ring_buf *buf, int data) {
    if (is_full(buf)) {
        return -1;
    }

    buf->data[buf->next_in] = data;
    buf->next_in = (buf->next_in + 1) % buf->size;

    return 0;
}
```

Listing 7.2: Programmcode des Erzeugers für den LockDoc-Test unter Linux.

```c
int is_empty(struct lockdoc_ring_buf *buf) {
     return buf->next_out == buf->next_in;
}


int consume(struct lockdoc_ring_buf *buf) {
    int result;

    if (is_empty(buf)) {
        return -1;
    }
    result = buf->data[buf->next_out];
    buf->next_out = (buf->next_out + 1) % buf->size;

    return result;
}
```

Listing 7.3: Programmcode des Verbrauchers für den LockDoc-Tests unter Linux.

- **Zu viele Sperren:** Alle vier Funktionen werden aufgerufen mit allen drei Sperren gehalten. Die Reihenfolge der Sperren aus Tabelle 7.1 wird dabei berücksichtigt.
- **Zu wenig Sperren:** Alle vier Funktionen werden, sofern möglich, mit weniger Sperren als erforderlich aufgerufen. Es wird allerdings mindestens eine Sperre gehalten. Daher wird für die Funktionen `is_full()` sowie `is_empty()` die korrekten Sperren-Regeln verwendet.
- **Keine Sperren:** Es werden alle vier Funktionen ohne den Einsatz von einer Sperre ausgeführt.
- **Falsche Reihenfolge:** Es werden lediglich die beiden Funktionen `produce()` und `consume()` ausgeführt. Dabei werden die korrekten Sperren eingesetzt, aber in der falschen Reihenfolge .

Auf Basis dieser Szenarien können nun die Zugriffsstatistiken der einzelnen Elemente der Datenstruktur gemäß des Schemas aus Kapitel 5 abgeleitet

Element	Zugriffstyp	Beobachtet	Aggregiert	WoR
`data`	r	0	0	0
	w	0	0	0
`next_in`	r	1	1	1
	w	0	0	0
`next_out`	r	1	1	1
	w	0	0	0
`size`	r	1	1	1

(a) Variablenzugriffe für die Datenstruktur `struct lockdoc_ring_buffer` in der Funktion `is_full()`.

Element	Zugriffstyp	Beobachtet	Aggregiert	WoR
`data`	r	0	0	0
	w	1	1	1
`next_in`	r	3	1	0
	w	1	1	1
`next_out`	r	1	1	1
	w	0	0	0
`size`	r	2	1	1

(b) Variablenzugriffe für die Datenstruktur `struct lockdoc_ring_buffer` in der Funktion `produce()` inklusive der Zugriffe innerhalb der Funktion `is_full()`.

Tabelle 7.2: Verteilung der Variablenzugriffe für die Datenstruktur `struct lockdoc_ring_buffer` aufgeteilt nach Element und Zugriffstyp. Die Zugriffe werden nach der Metrik aus Kapitel 5 in die Spalten *Beobachtet*, *Aggregiert* und *WoR* aufgelistet. Dargestellt sind Zugriffe für die Funktionen `is_full()` und `produce()`.

werden. Zur Verdeutlichung des Vorgehens sind in Tabelle 7.2a sowie Tabelle 7.2b exemplarisch die Zugriffe für die Funktionen `is_full()` und `produce()` für eine Ausführung dargestellt. In den Werten in Tabelle 7.2b sind bereits die Zugriffe für die Funktion `is_full()` berücksichtigt. Hier wird das Vorgehen von LockDoc noch einmal deutlich: Mehrfache Zugriffe innerhalb einer Transaktion – ein durch Sperren geschützter Bereich – zählen nur einfach. Dies wird anhand der Zahlen für das Lesen der Elemente `next_out` sowie `size` in Tabelle 7.2 deutlich. Die Spalte *WoR* ist hier der Vollständigkeit halber aufgelistet.

Unter Berücksichtigung der korrekt sowie falsch ausgeführten Funktionen ergibt sich die in Tabelle 7.3 aufgelisteten Variablenzugriffe für die jeweiligen Elemente. Bei Zugriffen, bei denen mindestens eine Sperre gehalten ist, werden die Zugriffe gemäß der Spalte *Aggregiert* gezählt. Bei Zugriffen gänzlich ohne Sperren wird indes die Spalte *Beobachtet* ausgewertet. Für das Lesen der Elemente `next_out` und `size` ohne Sperre innerhalb der Funktion

Element	Zugriffstyp	Total
`data`	r	204
	w	204
`next_in`	r	816
	w	204
`next_out`	r	816
	w	204
`size`	r	612

Tabelle 7.3: Alle Variablenzugriffe nach der Zählweise *aggregiert* für die Datenstruktur `struct lockdoc_ring_buffer` während des LockDoc-Tests.

`produce()` bedeutet dies, dass sie mit 3 bzw. 2 Zugriffen berücksichtigt werden.

Anhand der Zugriffsstatistiken für eine einzelne Ausführung soll die Zusammensetzung der Zahlen in Tabelle 7.3 exemplarisch für das Lesen des Elements `next_in` in Gleichung 7.1 berechnet werden:

$$
\begin{aligned}
s_a = & \underbrace{200 * (1 + 1 + 1 + 1)}_{\text{Korrekte Iterationen}} \\
& + \underbrace{1 * (1 + 1 + 1 + 1)}_{\text{Zu wenig Sperren}} \\
& + \underbrace{1 * (1 + 1 + 0 + 0)}_{\text{falsche Reihenfolge}} \\
& + \underbrace{1 * (1 + 1 + 1 + 1)}_{\text{zu viele Sperren}} \\
& + \underbrace{1 * (3 + 1 + 1 + 1)}_{\text{keine Sperren}} \\
& = 816
\end{aligned}
\tag{7.1}
$$

Jeder Summand in Gleichung 7.1 ist mit dem jeweiligen Szenario annotiert, aus dem sich die Zugriffe ergeben. Die Summanden in den Klammern stehen für die Zugriffe in den vier Funktionen. Die Reihenfolge der Werte wird wie folgt den Funktionen zugeordnet: `produce()`, `consume()`, `is_full()` und `is_empty()`. Die anderen Werte in Tabelle 7.3 lassen sich aus der vorgenannten Beschreibung des LockDoc-Tests analog dazu berechnen. Der in Gleichung 7.1 errechnete Werte für alle Zugriffe entspricht ebenfalls dem absoluten Support s_a für die Hypothese „keine Sperren nötig“. Für die neun verbleibenden Hypothesen für das Lesen des Elements `next_in` ist der absolute Support in Tabelle 7.4 gegeben. Als Gewinnerhypothese wird bei einem Wert von $t_{ac} = 98,5$ die Hypothese #1 ausgewählt, die zu der erdachten Sperren-Regel in Tabelle 7.1 passt. In Abschnitt 5.2.3 wurde der Einfluss der Auswahlstrategie für die Gewinnerhypothese auf das Ergebnis deutlich. Das

ID	Sperren-Hypothesen	s_a	s_r
#0	*Keine Sperre nötig*	816	100 %
#1	`rb_lock`	810	99,26 %
#2	`consumer_lock`	205	25,12 %
#3	`producer_lock`	205	25,12 %
#4	`consumer_lock`→`rb_lock`	204	25,00 %
#5	`producer_lock`→`rb_lock`	204	25,00 %
#6	`producer_lock`→`consumer_lock`→`rb_lock`	4	0,49 %
#7	`producer_lock`→`consumer_lock`	4	0,492 %
#8	`rb_lock`→`producer_lock`	1	0,12 %
#9	`rb_lock`→`consumer_lock`	1	0,49 %

Tabelle 7.4: Die Liste der Hypothesen für das Lesen des Elements `next_in` der Datenstruktur `struct lockdoc_ring_buffer`.

dort beschriebene Phänomen ist auch bei den Hypothesen für den LockDoc-Test zu beobachten. Dazu soll die Hypothesenliste für das Lesen des Elements `data` betrachtet werden. Die zugehörigen Hypothesen sind in Tabelle 7.5 aufgelistet. Die Hypothese #1 weist die meisten Zugriffe auf, da die zugehörige Sperre, abgesehen von dem Sonderfall, in dem keine Sperren verwendet werden, immer eingesetzt wird. Ähnlich verhält es sich für Hypothese #2: Die Sperre kommt bei allen korrekten Zugriffen und bei den Sonderfällen „zu viele Sperren“ sowie „falsche Reihenfolge“ vor. Beide Hypothesen entsprechen jedoch nicht der vorgesehenen Sperren-Regel aus Tabelle 7.1, obwohl sie den größten absoluten Support aufweisen. Die letztendliche Gewinnerhypothese #3 trifft bei mehr als den vorgenannten 200 korrekten Iterationen zu, da diese Sperren-Reihenfolge auch für den Sonderfall „zu viele Sperren“ zutrifft. Sie wird jedoch erst bei der in dieser Arbeit vorgeschlagenen Auswahlstrategie *Bottom Up* als Gewinner ausgewählt. Abschließend ist auf Hypothese #5 zu verweisen: Sie resultiert genau aus dem einen fehlerhaften Zugriff des Szenarios „falsche Reihenfolge“.

Das geschilderte Testszenario wurde sowohl im Linux- als auch im FreeBSD-Kern implementiert. Hierdurch soll die Infrastruktur zur Aufzeichnung wie auch die Nachverarbeitung getestet werden. Für den Linux-Kern konnte für einen Wert von $t_{ac} = 98,5$ gezeigt werden, dass sowohl die Zugriffsstatistiken als auch die Sperren-Hypothesen korrekt vorhergesagt wurden. Für FreeBSD konnten die Werte und Hypothesen bei einem Wert von $t_{ac} = 98,5$ ebenfalls korrekt prognostiziert werden.

Ebenso wurden die zuvor als falsch deklarierten Zugriffe als solche erkannt. Lediglich der Fall, in dem alle Sperren eingesetzt wurden, wurde nicht erkannt. Bei Betrachtung der Vorgehensweise zur Auswertung ist dies auch logisch: Da die korrekten Regeln `producer_lock` →`rb_lock` und `consumer_lock`

ID	Sperren-Hypothesen	s_a	s_r
#0	*Keine Sperre nötig*	204	100 %
#1	`rb_lock`	203	99,51 %
#2	`producer_lock`	202	99,02 %
#3	`producer_lock`→`rb_lock`	201	98,53 %
#4	`producer_lock`→`consumer_lock`→`rb_lock`	1	0,49 %
#5	`rb_lock`→ `producer_lock`	1	0,49 %
#6	`producer_lock`→`consumer_lock`	1	0,49 %
#7	`consumer_lock`→`rb_lock`	1	0,49 %
#8	`consumer_lock`	1	0,49 %

Tabelle 7.5: Die Liste der Hypothesen für das Schreiben des Elements `data` der Datenstruktur `struct lockdoc_ring_buffer`.

→`rb_lock` beide in der beobachteten Sperren-Reihenfolge `consumer_lock` →`producer_lock`→`rb_lock` enthalten sind, wird dies nicht als Fehler erkannt.

7.2 LINUX

Nachdem im vorangegangenen Abschnitt 7.1 mittels eines Test-Szenarios der LockDoc-Ansatz in zwei existierenden Betriebssystemkernen getestet wurde, soll in diesem Abschnitt das Sperren im Linux-Kern selbst untersucht werden. Zunächst wird der Ansatz zur Verbesserung der Arbeitslast aus Unterabschnitt 5.1.2 in Unterabschnitt 7.2.1 evaluiert. Im Weiteren werden die erforderlichen Anpassungen (Abschnitt 7.2.2) sowie die sich ergebenen Limitierungen vorgestellt. Darauf folgt die Evaluation der Auswahlstrategien in Unterabschnitt 7.2.3. Abschließend folgen die Ergebnisse der eigentlichen Untersuchung, Abschnitt 7.2.4, untergliedert nach den drei Rubriken aus Abschnitt 6.4. Abschnitt 7.2.5 fasst die Resultate noch einmal zusammen.

7.2.1 *Arbeitslast*

In diesem Abschnitt wird in Unterabschnitt 7.2.1.1 zunächst der Evaluationsaufbau dargelegt, gefolgt von den Ergebnissen in Unterabschnitt 7.2.1.2.

7.2.1.1 *Aufbau*

Die folgenden Experimente wurden mit einem x86-Linux-Kern (amd64) in Version 5.4 durchgeführt. Der Kern wurde ohne Modul-Unterstützung sowie mit einer minimal Konfiguration übersetzt: Lediglich die Netzwerkunterstützung sowie die absolut notwendigen Treiber für das Dateisystem und zur Ausführung einer paravirtualisierten *QEMU*-basierten virtuellen Maschine sind aktiviert [LTS20]. Zur Aufzeichnung der ausgeführten Basisblöcke wurde die bereits im Linux-Kern integrierte Funktion *KCOV* [Vyu16] aktiviert

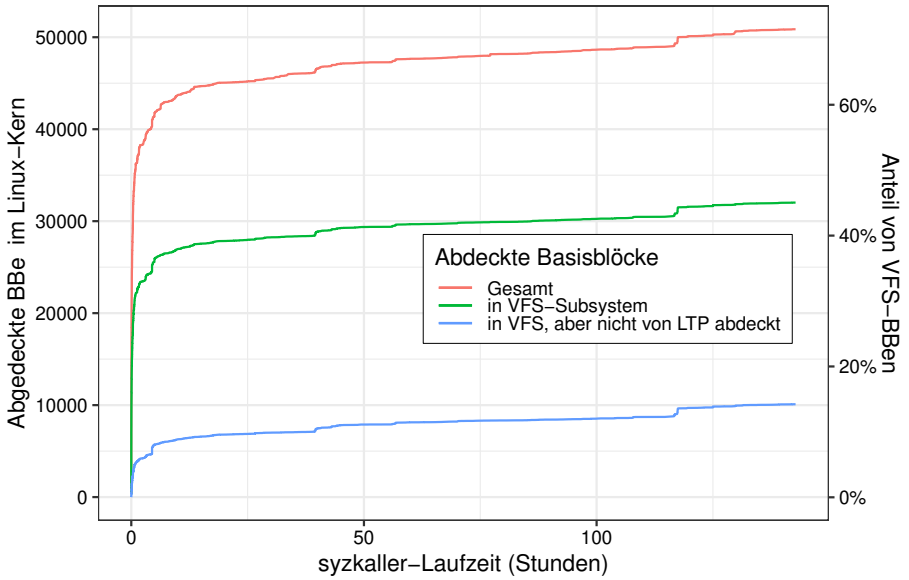


Abbildung 7.1: Entwicklung der Basisblockabdeckung der generierten Programme für den Linux-Kern über die Laufzeit von *syzkaller*. Die rote Linie stellt den Verlauf der abdeckten Basisblöcke an allen 300.860 Basisblöcken des Kerns dar. Die grüne Linie zeigt hingegen den Anteil an den Basisblöcken des VFS-Subsystems. Die blaue Linie bezeichnet den Anteil an neu abdeckten Basisblöcken, die noch nicht von LTP-Testsuites erreicht wurden. (Nach einer Vorlage von Lochmann et al. [LTS20])

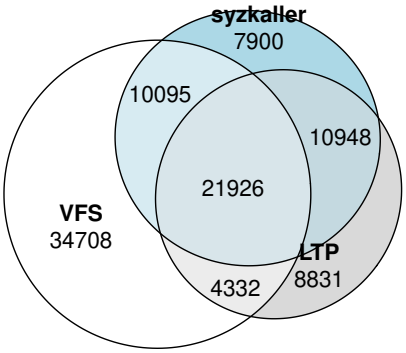
[LTS20]. Diese wurde initial mit der Entstehung von *syzkaller* in den Kern eingebracht [Vyu15]. Zusätzlich wurde noch die Erweiterung *KCOV_CMP*[2] aktiviert. Sie erlaubt es *syzkaller*, Operanden von Vergleichen in den *fuzzing*-Prozess einzubeziehen.

Die in Unterabschnitt 5.1.2.2 vorgestellten Modifikationen an *syzkaller* basieren auf Git-Commit *056be1b9c8d0c6942412dea4a4a104978a0a9311* [LTS20]. Insgesamt wurden 4 virtuelle Maschinen mit jeweils 2.048 MB Arbeitsspeicher und 2 CPU-Kernen ausgeführt. Innerhalb jeder virtuellen Maschine liefen 8 parallele Test-Prozesse. Außerdem wurde *syzkaller* nicht mit einem leeren Korpus an Programmen gestartet. Stattdessen wurde mit Hilfe der Arbeit von Pailoor et al. ein initialer Korpus erzeugt [PAJ18]. Dieser enthält bereits die wichtigsten Systemaufrufe im Sinne der Codeabdeckung [PAJ18]. Hierzu wurde die LTP-Testsuite *fs* mit den Werkzeugen von Pailoor et al. ausgewertet. Lediglich vier der insgesamt 66 Tests der Testsuite *fs* konnten nicht verarbeitet werden. Dieser Schritt sorgt dafür, dass *syzkaller* bereits komplexere Zusammenhänge zwischen Systemaufrufen bekannt sind und somit eine größere Codeabdeckung erzielen kann [PAJ18].

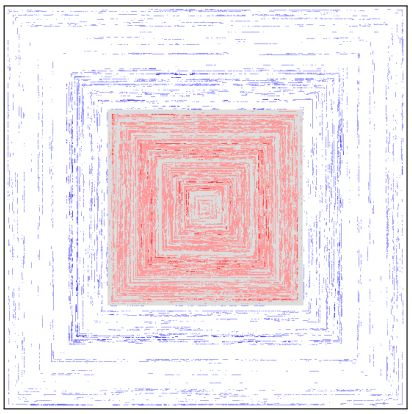
Basierend auf einem Beispielprogramm von *syzkaller* wurde eine Bibliothek in 465 Zeilen[3] *C++*-Code realisiert, die zur Aufzeichnung der ausgeführten Basisblöcke von beliebigen Programmen eingesetzt werden kann

2 https://www.kernel.org/doc/html/v5.4/dev-tools/kcov.html

3 Anzahl der effektiven Zeilen Code, ohne Leerzeilen und Kommentare. Ermittelt mittels *cloc* (https://github.com/AlDanial/cloc).



(a) Mengenbeziehungen zwischen den Basisblöcken im VFS-Subsystem und den durch *syzkaller*-Programme sowie LTP-Testsuites ausgeführten Basisblöcken. Die Nummern stellen die absolute Anzahl an Basisblöcken in den jeweiligen Mengen dar. (Nach einer Vorlage von Lochmann et al. [LTS20])



(b) Basisblockabdeckung aller von *syzkaller* generierten Programme. Die graue Box grenzt die Basisblöcke des VFS-Subsystems ab. In Rot dargestellt sind die im VFS-Subsystem abgedeckten Basisblöcke, blau bezeichnet die im restlichen Kern ausgeführten Basisblöcke.

Abbildung 7.2: Darstellungen zur Mengenbeziehung zwischen allen Basisblöcken und den im VFS-Subsystem ausgeführten Basisblöcken.

[LTS20]. Es können ganze Prozessbäume aufgezeichnet werden, da sich die genannte Bibliothek mit Hilfe des *LD_PRELOAD*-Mechanismus in das zu untersuchende Programm einklinkt [LTS20]. Dies kam zum Einsatz, um die in Unterabschnitt 5.1.2 vorgestellten Ergebnisse zu erheben [LTS20].

Um zu bestimmen zu welchen Subsystem ein Basisblock gehört, wird die Adresse des Basisblocks mit Hilfe des Werkzeugs *addr2line*[4] sowie dem *ELF*-Abbild des Linux-Kerns in eine konkrete Stelle innerhalb einer Quellcodedatei übersetzt [LTS20]. Durch den Vorgang des *function inlining* kann eine Basisblockadresse zu mehreren Quellcodedateien übersetzt werden [LTS20]. Wenn eine der ermittelten Dateien dem folgenden regulären Ausdruck genügt, wird der zugehörige Basisblock als zum VFS-Subsystem-zugehörig gewertet: */fs/|/mm/|fs.h|mm.h*. Das Verzeichnis `mm` wie auch die Header-Dateien, die `mm.h` beinhalten, werden ebenfalls berücksichtigt, da sie für die Datei-Ein-und-Ausgabe relevanten Code, wie z. B. `mm/readahead.c` oder `mm/page-writeback.c`, beinhalten [LTS20].

7.2.1.2 *Ergebnisse*

Die modifizierte Variante von *syzkaller* lief für 142,76 Stunden und erzeugte dabei 3.288 Programme. Diese führten zu einer Basisblockabdeckung von 16,91 % für den gesamten Linux-Kern sowie zu 45,06 % Abdeckung innerhalb des VFS-Subsystems. Viel wichtiger ist, dass dabei 14,21 % von den Basisblöcken innerhalb des VFS-Subsystem erreicht wurden, die vorher nicht bereits von LTP-Testsuites abgedeckt wurden.

4 https://sourceware.org/binutils/docs/binutils/addr2line.html

Die Abbildung 7.1 stellt den zeitlichen Verlauf dieser drei Werte über den kompletten Lauf von 142,76 Stunden dar. Die Basisblockabdeckung steigt zu Beginn des Laufs schnell an und schwächt sich danach ab. Die linke Y-Achse stellt die absolute Anzahl an ausgeführten Basisblöcken dar. Wohingegen die rechte Y-Achse den Anteil an Basisblöcken vom VFS-Subsystem angibt. In Abbildung 7.2b wird die Basisblockabdeckung am Ende des Laufs dargestellt. Die Abbildung stellt die räumliche Verteilung der Basisblöcke dar. Wobei die graue Box alle Basisblöcke des VFS-Subsystems bezeichnet.

Die Relation zwischen der Codeabdeckung, den generierten *syzkaller*-Programmen sowie den LTP-Testsuites ist in Abbildung 7.2a nochmal als Venn-Diagramm visualisiert. Die Größe der Flächen ist proportional zu der Anzahl der Basisblöcke. Die Nummern in jeder Fläche stellen die absolute Anzahl an Basisblöcken dar, die die durch die geschnittenen Flächen repräsentierten Basisblockmengen gemein haben.

Zusammenfassend ist festzuhalten, dass die Kombination aus LTP-Testsuites und von *syzkaller* generierten Programmen die Basisblockabdeckungen um 14,21 Prozentpunkte von 36,95 % auf 51,16 % steigern kann. Dies kommt einer Steigerung der Basisblockabdeckung von 38,45 % gleich.

7.2.2 *Anpassungen an das Betriebssystem*

Für die Untersuchungen im Rahmen dieser Arbeit wurde der Linux-Kern in Version 5.4 (i386) als Zielsystem ausgewählt. Dabei werden von den verschiedenen Synchronisationsmechanismen des Linux-Kerns [BC05] insgesamt 6 Sperren-Typen unterstützt. Hierzu zählen `spinlock_t`, `rw_lock_t`, `semaphore`, `rcu`, `mutex` und `rw_semaphore` [LSBS19]. Diese Sperren-Typen folgen alle dem Modell aus Abbildung 5.1 und dienen dem Sicherstellen des (geteilten) gegenseitigem Ausschlusses. Der Sperren-Typ *Sequential Locks*[5] (`seqcount_t` und `seqlock_t`) wird nicht unterstützt, da er keine Atomizität innerhalb des kritischen Abschnitts auf der Leser-Seite gewährleistet [Lov10]. Erst beim Verlassen des kritischen Abschnitts durch den Leser wird überprüft, ob ein exklusiver Zugriff bestand [Lov10]. Dies widerspricht jedoch dem Sperren-Modell, das LockDoc zu Grunde liegt. Xu et al. umgehen dieses Problem beispielsweise, indem sie ihre Aufzeichnungen vollständig einlesen, um die genannten Überprüfungen zu finden [XKZK20]. Erst dann führen sie offline ihre Analyse durch [XKZK20]. Es bleibt zukünftigen Arbeiten überlassen, hier eine akzeptable Lösung zu finden. Tabelle 7.6 gibt nochmal eine Übersicht über alle unterstützten Synchronisationsmechanismen sowie deren Abbildung auf das Sperren-Modell aus Abschnitt 6.1.

Die Aufzeichnung der Sperren-Operationen erfolgt über die Instrumentierung der Programmierschnittstellen der jeweiligen Sperren-Typen. Neben der Operation und der beteiligten Sperren werden auch Kontextinformationen, wie die Stelle im Programmcode, die aufrufende Funktion und die verwendete Funktion der Programmierschnittstelle, aufgezeichnet [LSBS19]. Zusätzlich werden die synthetischen Sperren *softirq* und *hardirq* [BC05] ab-

5 https://www.kernel.org/doc/html/latest/locking/seqlock.html

Modell	Sperren-Typen
Leser-Sperre	`rcu`
Schreiber-Sperre	`spinlock_t`, `semaphore`, `mutex`, `hardirq`, `softirq`, Präemption
Leser-Schreiber-Sperre	`rw_lock_t`, `rw_semaphore`
Nicht abgebildet	`struct completion`, `seqlock_t`, `seqcount_t`, Barrieren, atomare Operationen

Tabelle 7.6: Eine Übersicht über die von LockDoc unterstützten Synchronisationsmechanismen in Linux. Jeder Mechanismus wird einem Teil des Modells aus Abbildung 5.1 zugeordnet. Die Basis für die Auflistung stellt die Übersicht von Robert Love über Synchronisationsmittel dar [Lov10].

gebildet [LSBS19]. Sie bilden das An- und Abschalten der Unterbrechungen sowie der nachgelagerten Unterbrechungsbehandlung ab.

Damit Optimierungen, wie z. B. das Entfernen von Sperren[6], ausgeschlossen werden, wird der Linux-Kern mit der Funktion *Symmetric multiprocessing* (kurz: SMP) konfiguriert [LSBS19]. Allerdings existiert die Präemptionssperre nicht, da die Konfigurationsoption `CONFIG_PREEMPT` nicht gesetzt ist.

Im *Virtual Filesystem* (VFS) werden insgesamt 17 Datentypen untersucht: `struct super_block`, `struct backing_dev_info`, `struct block_device`, `struct pipe_inode_info`, `struct cdev`, `struct address_space`, `struct bdi_writeback`, `struct journal_t`, `struct transaction_t`, `struct journal_head`, `struct dentry`, `struct dentry_aux`, `struct lockref`, `struct qstr`, `struct buffer_head`, `struct lockdoc_ring_buffer`, `struct jbd2_inode`[LSBS19]. In Unterabschnitt 7.2.4.2 wird jedoch nur eine Teilmenge dieser Datentypen aufgelistet, da Datentypen im Linux-Kern nicht selten in einander eingebettet sind. Listing 7.4 zeigt dies anhand des Beispiels für die Datenstrukturen `struct inode` und `struct adress_space`. Beide Datenstrukturen werden in der zuvor genannten Auflistung geführt. In Unterabschnitt 7.2.4.2 findet sich jedoch lediglich die Datenstruktur `struct inode` wieder. Da die Struktur `struct address_space` jedoch als relevant für das VFS-Subsystem angesehen wird, sie aber in die Struktur `struct inode` eingebettet ist, wird sie ebenfalls aufgeführt. Wäre dies nicht der Fall, würde LockDoc lediglich Sperren-Regeln für das Element `i_data` bestimmen – vgl. Listing 7.4. Zur Aufzeichnung der Allokations- und Freigabeoperationen einer Datenstrukturen wurden die zugehörigen Schnittstellen im Kern für die jeweilige Datenstruktur, wie z. B. die Funktion `alloc_inode()`, instrumentiert. Neben der Startadresse des Objekts wird die Größe und der Datentyp aufgezeichnet [LSBS19]. Für die Datenstruktur `struct inode` wird ebenfalls der dahinter liegende Dateisystemtyp, die Unterklasse, aufgezeichnet. Für jeden Datentyp müssen die Funktionen zur Initialisierung sowie zum Aufräumen eines Objektes ermittelt und in der entsprechenden Liste vermerkt werden. Dieser Prozess muss ggf. wiederholt werden, wenn sich beim Betrachten der ersten

6 https://www.kernel.org/doc/html/v5.4/kernel-hacking/locking.html?highlight=uniprocessor#locks-and-uniprocessor-kernels

```
1  struct address_space {
2      struct inode        *host;
3      struct xarray       i_pages;
4      gfp_t               gfp_mask;
5      // ...
6  };
7
8  struct inode {
9      // ...
10     struct address_space i_data;
11     // ...
12     union {
13         struct pipe_inode_info    *i_pipe;
14         struct block_device       *i_bdev;
15         struct cdev               *i_cdev;
16         char                      *i_link;
17         unsigned                  i_dir_seq;
18     };
19 };
```

Listing 7.4: Beispiel für verschachtelte Datentypen sowie das Ausrollen eines Unions in Linux (Entnommen aus `linux-v5.6/include/linux/fs.h`, Zeilen 445 und 628).

Gegenbeispiele herausstellt, dass die Listen noch unvollständig sind. Für Linux wurden in Summe 387 Funktionen gefiltert. Ebenso werden bestimmte Elemente einer Datenstruktur, wie z. B. die Sperren selbst, gefiltert. Hierzu zählen aber auch Elemente, wie beispielsweise `inode.i_dir_seq` oder `inode.i_data.i_pages`, die ihr eigenes Vorgehen beim Sperren wählen. In Linux werden in Summe 41 Elemente gefiltert.

Im Linux-Kern werden Elemente einer Datenstruktur in Form des *C*-Konstrukts `union` zusammengefasst, um Speicher zu sparen [LSBS19]. Ein Beispiel dafür ist ebenfalls in Listing 7.4 in Zeile 12 ff. gegeben. Da diese Elemente jedoch alle denselben Offset innerhalb der Datenstruktur aufweisen, können sie von LockDoc nicht unterschieden werden [LSBS19]. Daher findet in solchen Fällen ein sog. „Unrolling“ statt, so dass jedes Element einen eigenen, eindeutigen Offset erhält [LSBS19]. In dem Fall aus Listing 7.4 werden die betreffenden Elemente schlicht nicht mehr in einem `union` zusammengefasst.

7.2.3 *Auswahlstrategien*

Die Erläuterungen der einzelnen Auswahlstrategien anhand des Beispiels aus Abschnitt 5.2.3 machten bereits deutlich, dass die Ergebnisse mit den Parametern für die jeweiligen Verfahren schwanken können. Daher werden in diesem Abschnitt sowohl der Einfluss der Parameterwahl auf die einzelnen Verfahren als auch die Strategien miteinander verglichen. Neben dem Vergleich der Auswahlstrategien werden gleichzeitig die unterschiedlichen Arten der Aufbereitung der Daten untersucht. Aus den Überlegungen aus Abschnitt 5.2.2 bzw. Abschnitt 6.2 ergeben sich die folgenden vier Kombinatio-

nen: kontext-sensitiv+*aggregiert*, kontext-sensitiv+*WoR*, kontext-insensitiv-+*aggregiert* sowie kontext-insensitiv+*WoR*. Die Begriffe *aggregiert* und *WoR* bezeichnen die Zählweise der Speicherzugriffe aus Abschnitt 5.2.2.

Algorithmus 7.1 : Berechnung des Anteils der korrekt vorhergesagten Sperren-Regeln f_h

Datenstrukturen : Jede Map ordnet einem Tupel aus Datentyp, Element und Zugriffstyp einer Sperren-Regel zu.
Eingabe : G ist eine Map der dokumentierten Sperren-Regeln,
H ist eine Map *aller* beobachteten Hypothesen,
I ist eine Map aller als Gewinner ausgewählten Hypothesen.
Ausgabe : f_h

```
1  Beginn
2      b = 0 ; /* Zähler für die beobachteten Tupel          */
3      m = 0 ; /* Zähler für die korrekt vorhergesagten Tupel
           */
4      für jedes Tupel t mit der Sperren-Regel r aus G tue
5          wenn t in I existiert dann
6              b = b + 1;
7              Sei s die Gewinnerhypothese zu t aus I;
8              wenn r == s dann
9                  m = m + 1;
10         sonst wenn t in H existiert dann
11             b = b + 1;
12     f_h = m/b
```

Als Vergleichsmetrik wird der Anteil korrekt vorhergesagten Sperren-Regeln f_h herangezogen. Die Berechnung von f_h ist in Algorithmus 7.1 dargestellt. Hierzu werden die zentral dokumentierten Datenstrukturen aus dem Linux- und FreeBSD-Kern herangezogen. Genauere Informationen dazu finden sich dazu in Unterabschnitt 7.2.4.2 bzw. 7.3.3.2. Die Tabelle 7.7 stellt die Statistik zu den untersuchten Sperren-Regeln für Linux und FreeBSD dar. Für einen besseren Vergleich zwischen Linux und FreeBSD werden die Zahlen für beide Betriebssystemkerne hier aufgeführt. Die Evaluation der Auswahlstrategien für FreeBSD findet sich in Unterabschnitt 7.3.2. Die Anzahl der dokumentierten Regeln für Linux und FreeBSD beträgt 172 bzw. 200. Eine Regel definiert die benötigten Sperren inkl. der Reihenfolge für ein Tupel aus Datenstruktur, Element und Zugriffstyp. Da zwischen lesenden und schreibenden Zugriffen differenziert wird, wurde die Dokumentation in Linux von 86 Elementen von 8 verschiedenen Datenstrukturen herangezogen. In der Aufzeichnung finden sich insgesamt Daten für 151 Tupel unter Linux und für 174 Tupel unter FreeBSD. Als Arbeitslast wird für Linux die Testsuite *syscalls* aus dem LTP-Projekt verwendet. Für FreeBSD kommt hingegen die Testsuite *fs* zum Einsatz. Für die Strategien *Top Down* und *Bottom Up* wurden jeweils Akzeptanzschwellenwerte von $0,9 \leq t_{ac} \leq 1$ untersucht. Für die Strategie *Top Down* wurde $n_{nolock} = 0,05$ verwendet. Analog zu der Wahl des Akzeptanzschwellenwertes wird für die Strategie *Sharpen* $0,0 \leq t_s \leq 0,1$

Betriebssystem	#Regeln	#Elemente	#Datenstrukturen	#Beobachtet
Linux	172	86	8	151
FreeBSD	200	100	3	174

Tabelle 7.7: Zusammenfassung der verwendeten Sperren-Regeln: Für Linux und FreeBSD wird die Anzahl dokumentierter Regeln (r+w), die dokumentierten Elemente und Datenstrukturen sowie die Anzahl der beobachteten Tupel aus Datenstruktur, Element und Zugriffstyp ausgewiesen.

untersucht. Aufgrund der Struktur der *LockSet*-Strategie gibt es hier keine Parameter zu evaluieren.

Auf Basis dieser Parameterwertebereiche ist für den Verlauf des Anteils der korrekt vorhergesagten Sperren-Regeln Folgendes zu erwarten:

1. Für eine großzügige Wahl von t_{ac} bzw. t_s fällt f_h gering aus, da Hypothesen akzeptiert werden, die möglicherweise zufällig noch gehaltene Sperren enthalten. Hier ist der relative Support der Gewinnerhypothese zu niedrig.

2. Mit steigenden Werten t_{ac}und t_s wird f_h ebenfalls bis zu einem bestimmten Maximum ansteigen, da sich nun die Hypothesen mit den korrekten Sperren, gemäß Dokumentation, durchsetzen.

3. Für eine restriktive Parameterwahl mit $t_{ac} = 1,0$ und $t_s = 0,0$ ist zu erwarten, dass f_h deutlich abfällt, da hier nur Hypothesen mit $s_r = 1$ akzeptiert werden. Nicht jede der richtigen Hypothesen weist aber einen relativen Support von 100 % auf. Wie im Verlauf dieser Arbeit aber noch in u.a. Abschnitt 7.2.4 detaillierter erläutert wird, hängt dies mit der Struktur von LockDoc zusammen: Abseits von Programmcode zum Erzeugen oder Zerstören von Objekten gibt es Stellen in Betriebssystemen, an denen keine Synchronisation erforderlich ist, da z. B. die Datentypen atomar zugreifbar sind oder es aber keine Nebenläufigkeit gibt und dies noch nicht in LockDoc korrekt abgebildet ist. Außerdem kann an der betreffenden Stelle im Programmcode die Konsistenz eine untergeordnete Rolle spielen, so dass auf eine Synchronisation verzichtet wird. Dies gilt insbesondere für den Linux-Kern.

In Abbildung 7.3 ist der Verlauf des Anteils der korrekt vorhergesagten Sperren–Regeln f_h gruppiert nach der jeweiligen Strategie für den Linux-Kern abgebildet. Auf der x-Achse sind die Wertebereich der beiden Parameter t_{ac} und t_s aufgetragen. Grundsätzlich fällt bei allen drei Auswahlstrategien sofort auf, dass die Ergebnisse basierend auf der Zählung *aggregiert* besser ausfallen als die basierend auf der Zählung *WoR*. Eine Stichprobe der Ergebnisse für die Strategie *Bottom Up* ergab, dass das Ignorieren der Lesezugriffe innerhalb von schreibenden Transaktionen zu einem geringen relativen Support der – laut Dokumentation – korrekten Hypothese führte. Sie ging im Rauschen unter. Es gab außerdem Tupel, für die mittels der Zählung *aggregiert* im Vergleich zu *WoR* überhaupt Sperren-Regeln abgeleitet werden konnte,

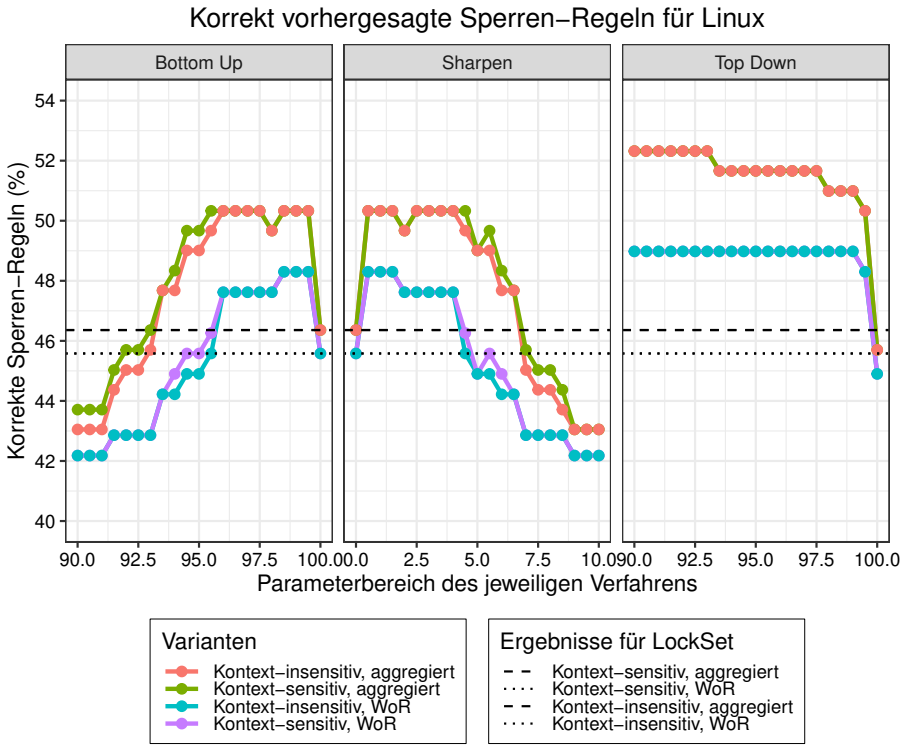


Abbildung 7.3: Auswirkungen der Auswahlstrategien und deren Parameter auf die Sperren-Regeln für den Linux-Kern: Die y-Achse stellt den Anteil der korrekt vorhergesagten Sperren-Regeln f_h dar. Die x-Achse bildet die untersuchten Parameterwerte ab. Für die Strategien *Top Down* und *Bottom Up* werden Werte im Bereich von $0,9 \leq t_{ac} \leq 1$ gewählt. Für die Strategie *Sharpen* hingegen gilt $0,0 \leq t_s \leq 0,1$. Zum Vergleich stellen die verschiedenen, schwarzen Linien die Ergebnisse der *LockSet*-Strategie je nach Eingabedaten dar.

da es nun Beobachtungen gab. Dies betraf meist das Lesen von Elementen. Es ist zu vermuten, dass die zugrundeliegenden Zugriffe alle in schreibenden Transaktionen stattfanden. In Summe stützt dies die Argumentation aus Abschnitt 5.2.2.

Auffällig sind die im Vergleich zu den anderen Strategien schlechteren Ergebnisse der *LockSet*-Strategie. Da diese nur Hypothesen mit einem relativen Support von 100 % akzeptiert, müssen die korrekt vorhergesagten Hypothesen, die für den Abstand zwischen den Ergebnissen für die *LockSet*-Strategie und der Resultate der anderen Strategien sorgen, einen relativen Support unter 100 % aufweisen. Dies kann zweierlei Gründe haben: a) Die Filterung umfasst noch nicht alle Kontexte für die Initialisierung und Zerstörung von Objekten. b) Es gibt valide Zugriffe, die die Sperren-Regeln umgehen. Zur Klärung bedarf es jedoch einer Rücksprache mit den Linux-Entwicklern.

Die im vorangegangenen Abschnitt aufgestellten Annahmen zum Verlauf von f_h lassen sich für die Strategien *Bottom Up* und *Sharpen* in der Abbildung wiederfinden. Maximal erreichen beide Strategien einen Anteil von 50,33 %. Dabei reichen die Schwellenwerte t_{ac} bzw. t_s dafür von 98,5 % - 99,5 % bzw. von 0,5 % - 1,5 %. Die Strategien *Bottom Up* und *Sharpen* erzielen, wie in

Abbildung 7.3 zu sehen ist, für $t_{ac} = 1,0$ und $t_s = 0,0$ das gleiche Ergebnis wie die *LockSet*-Strategie. Dies in den Parameterwerten für *Bottom Up* und *Sharpen* begründet: Für die vorgenannten Werte verhalten sich die Strategien wie die *LockSet*-Strategie, da sie nur Sperren-Regeln akzeptieren, die einen relativen Support von 100 % aufweisen. Ebenfalls auffällig ist, dass a) die Kurve für die Variante kontext-sensitiv+*aggregiert* minimal besser ist als die Variante kontext-insensitiv+*aggregiert* und b) sowohl *Bottom Up* als auch *Sharpen* einen Knick bei $t_{ac} = 0,98$ respektive $t_s = 0,02$ aufweisen.

Zunächst soll exemplarisch anhand der Ergebnisse der Strategie *Bottom Up* mit dem Akzeptanzwerten von $t_{ac} = 0,95$ sowie $t_{ac} = 0,925$ geschaut werden, warum die Resultate bei einer kontext-sensitiven Nachverarbeitung besser ausfallen. In beiden Fällen ist für die kontext-sensitive Variante jeweils die Regel für genau ein weiteres Tupel korrekt vorhergesagt worden. In der Variante kontext-insensitiv hatte eine andere, falsche Hypothese einen minimal besseren relativen Support, der oberhalb des Akzeptanzschwellenwertes lag. Da die Strategie *Bottom Up* die Liste der Hypothesen von unten betrachtet, gewinnt in diesem Fall die falsche Hypothese. Bei der kontext-sensitiven Variante hat ebendiese Hypothese einen relativen Support unterhalb des Akzeptanzschwellenwertes und gewinnt somit nicht. Berücksichtigt man die geringen Abstand zwischen den Kurven bleibt es aber fraglich, ob sich dieser Unterschied bei jedem Datensatz zeigt. Abseits davon wird für die Analyse dieses Datensatzes in Abschnitt 7.2.4 die kontext-sensitive Nachverarbeitung verwendet.

Für die zweite Besonderheit (b) werden die korrekt vorhergesagten Regeln für die Akzeptanzschwellenwerte von 97,5 % und 98,0 % verglichen. Für ein Tupel wechselt hierbei die Vorhersage von korrekt auf falsch. Für dieses Tupel gewinnt die leere Hypothese, da die eigentlich korrekte Hypothese einen relativen Support von 97,9 % aufweist und somit abgelehnt wird. Im nächsten Schritt von 98 % auf 98,5 % Akzeptanzschwellenwert wird für ein anderes Tupel eine korrekte Hypothesen vorhergesagt, so dass insgesamt wieder ein Anteil von 50,33 % erreicht wird. Exakt das gleiche Szenario spielt sich bei den Ergebnissen für die Strategie *Sharpen* für die Werte von 2,5 % bis 1,5 % ab. Es bleibt zukünftigen Arbeiten überlassen zu überprüfen, wie sich die Menge der Tupel, für die eine korrekte Sperren-Regel vorhergesagt wurde, über den Parameterraum entwickelt. Aufgrund der vorgenannten Beobachtung ist zu vermuten, dass sie nicht monoton wachsend ist. Stattdessen ist anzunehmen, dass vereinzelt Tupel wegfallen und dafür neue Tupel hinzukommen.

Der Verlauf der Ergebnisse für die Strategie *Top Down* entspricht nicht den Annahmen 1. und 2. Die Ergebnisse fallen hier mit wachsendem Schwellwert ab. Der maximal Wert von 52,32 % ist bereits direkt zu Beginn bei einem Schwellenwert von 90 % erreicht. Gleichzeitig zeigt die Strategie *Top Down* im Bereich von 90 % - 93 % das beste Ergebnis von allen drei Strategien. Beim exemplarischen Vergleich der Strategien *Top Down* und *Bottom Up* für die Akzeptanzschwellenwerte von 90,0 % respektive 98,5 % stellte sich heraus, dass erstere in vier zusätzlichen Fällen die Regeln korrekt vorhersagt. Allerdings

versagt die Strategie *Top Down* auch in einem Fall im Vergleich zu *Bottom Up*. Hier lag die Anzahl Speicherzugriffe ohne Sperren unterhalb des Schwellenwertes $n_{nolock} = 0,05$. Von den vier zusätzlichen korrekten Sperren-Regeln ist in zwei Fällen der relative Support unterhalb des Akzeptanzschwellenwertes von 98,5 %, so dass sie von der Strategie *Bottom Up* nicht ausgewählt werden. In den anderen beiden Fällen ist der Akzeptanzschwellenwert für die Strategie *Bottom Up* hingegen zu niedrig gewählt. Hier werden zwei Hypothesen ausgewählt, bei denen gilt $t_{ac} \leq s_r \leq 1$. Die korrekten Hypothesen haben jedoch einen relativen Support von 100 %.

Lediglich die dritte Annahme ist auch hier zu beobachten. Auffällig ist allerdings, dass das Ergebnis für $t_{ac} = 1,0$ unter das Ergebnis der *LockSet*-Strategie von 46,36 % fällt. Im Gegensatz zu den anderen Strategien, die immer eine Gewinnerhypothese bestimmen, kann die *Top Down*-Strategie für ein Tupel kein Ergebnis liefern. Bei der manuellen Untersuchung der Ergebnisse war dies für ein Tupel zu beobachten.

Auffällig ist, dass keine der Strategien letztlich über 52,32 % hinaus kommt. Wenngleich die Strategie *Top Down* die besten Ergebnisse zeigt, konnte bereits in Abschnitt 5.2.3 dargelegt werden, warum sie als Auswahlstrategie weniger geeignet ist. Im weiteren Verlauf dieser Arbeit kommt deshalb die Strategie *Bottom Up* zum Einsatz.

Außerdem wird für diese Strategie eine manuelle Untersuchung der Ergebnisse für $t_{ac} = 0,985$ für die Variante kontext-sensitiv+*aggregiert* vorgenommen. Die Gründe, warum nicht die korrekten Hypothesen gewinnen, sind verschieden: Die Tabelle 7.8 liefert hierzu eine Aufschlüsselung. Die Kategorie „fehlerhafte Dokumentation“ fasst alle falschen Regeln zusammen, wo sich möglicherweise eine falsche Dokumentation hinter verbirgt. In zwei Fällen, ein Element betreffend, führten die Ergebnisse bereits zu einer akzeptierten Änderung[7]. In den verbleibenden fünf Fällen, drei Elemente betreffend, wurde ein Änderungsvorschlag eingereicht und für gut befunden[8]. Allerdings ist hier anzumerken, dass mit einer aktualisierten Dokumentation dennoch nicht die richtigen Regeln gewännen, da entweder zu viele Sperren beteiligt sind oder aber der Akzeptanzschwellenwert zu niedrig gewählt ist. Die Gruppe „Strategie versagt“ fasst Regeln zusammen, bei denen die *Bottom Up*-Strategie mit $t_{ac} = 0,985$ versagt hat. Die korrekte Hypothese hatte zumeist einen größeren relativen Support wurde aber aufgrund der Parameterwahl nicht ausgewählt. In anderen Situationen hingegen ist bei jedem Zugriff auf eine Datenstruktur neben den richtigen Sperren ebenfalls noch mindestens eine weitere Sperre beteiligt, wie z. B. $a \rightarrow b \rightarrow c$ statt nur $b \rightarrow c$. Diese hat der Dokumentation nach zu urteilen nicht unmittelbar etwas mit der zugegriffenen Datenstruktur zu tun. Dennoch gewinnt die Hypothese $a \rightarrow b \rightarrow c$, da sie bei gleichem relativen Support mehr Sperren beinhaltet. Solche Situation werden unter dem Punkt „zu viele Sperren“ aufsummiert. Die Kategorie „zu wenig rel. Support“ fasst diejenigen Fälle zusammen, bei denen die korrekte Hypothese einen zu geringen relativen Support aufweist.

7 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=50a4952fd67b7f7f551e82ac07c51c1a7a74d474

8 https://lkml.org/lkml/2021/3/19/313

Fehlertyp	Häufigkeit
fehlerhafte Dokumentation	7
Strategie versagt	3
zu viele Sperren	19
zu wenig rel. Support	39
sonstige	7

Tabelle 7.8: Aufschlüsselung der Gründe sowie deren Häufigkeit für falsch vorhergesagte Sperren-Regeln für die Strategie *Bottom Up* für $t_{ac} = 0,985$. Es wurden die falsch vorhergesagten Sperren-Regeln für 75 Tupel aus Datenstruktur, Element und Zugriffstyp ausgewertet.

Hierbei ist der Abstand zwischen dem relativen Support der korrekten Hypothese und der Gewinnerhypothese zu groß, als dass er über eine Justage des Parameters abgefangen werden könnte.

Von den genannten 39 Fällen entfallen 31 auf das Lesen von Elementen. Im Austausch mit den Kern-Entwicklern wurde deutlich, dass Datentypen mit Wortgröße, wie z. B. `int`, aus Performanzgründen als atomar lesbar angenommen werden. Entweder muss an dieser Stelle der LockDoc-Ansatz oder die Dokumentation im Linux-Kern angepasst werden. Für die Datenstrukturen `struct journal_s` und `struct transaction_s` wurden bei den betreffenden Entwicklern nachgefragt, ob sich die Dokumentation hier ggf. aktualisieren ließ, so dass zwischen lesendem und schreibendem Zugriff unterschieden wird. Aus einer längeren Diskussion[9] mit einem Entwickler entstanden zwei Änderungen, die auch bereits final akzeptiert wurden[10]. Beide Änderungen ergänzen die Dokumentation um Randfälle, bei denen keine Sperren erforderlich sind. Jan Kara bestätigte auf Nachfrage, dass der sich ausgeführte Programmcode in einem der beiden Fälle im wesentlichen in einer Funktion befindet[11]. Durch das Filtern dieser Zugriffe können die Ergebnisse von LockDoc in Zukunft noch verbessert werden.

Für die Datenstruktur `struct dentry` weisen die Hypothesen in vielen Fällen einen zu geringen Support auf bzw. die vermeintlich korrekten Hypothesen tauchen gar nicht auf. Insgesamt werden wenig Regeln korrekt vorhergesagt. Daher liegt die Vermutung nahe, dass LockDoc diese Fälle noch nicht korrekt abbilden kann. Hierfür spricht auch die bereits erwähnte Dokumentation zum anwendungsspezifischen Sperren innerhalb von Dateisystemen[12]. Es werden teilweise Sperren aus verschiedenen Instanzen benötigt, die aber in Beziehung zu einander stehen.

9 https://www.spinics.net/lists/linux-ext4/msg74532.html
10 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d699ae4fc27496d01e8bc5ab2106bd79d1e7be92
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3042b1b45c4106feff063932d4fd481c5009dbe1
11 https://lkml.org/lkml/2021/3/29/1177
12 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/locking.rst?h=v5.6

Abschließend ist festzuhalten, dass die in Kapitel 6 vorgestellte Heuristik *Bottom Up*, neben der Strategie *Sharpen*, nicht die Resultate der Strategie *Top Down* erreicht. Da sie aber, wie bereits in Abschnitt 5.2.3 erläutert, anfällig für die Auswahl der falschen Hypothese ist, wird im weiteren Verlauf der Arbeit auf die Strategie *Bottom Up* gesetzt. Wenngleich die Ergebnisse aus Abbildung 7.3 den Schluss zulassen, dass es einen optimalen Parameterbereich von 98,5 % bis 99,5 % gibt, so wird im Folgenden für den Schwellwert t_{ac} die untere Grenze gewählt. So bleibt genug Spielraum, damit valide Zugriffe, die die Sperren-Regeln absichtlich umgehen, nicht zu einer falschen Gewinnerhypothese führen.

7.2.4 *LockDoc-Analyse*

In diesem Abschnitt wird der Aufbau zur Datenerhebung unter Linux, Unterabschnitt 7.2.4.1, erklärt. Diesem folgt eine Auswertung der verarbeiteten Daten nach den in Abschnitt 6.4 vorgestellten Kategorien in den darauf folgenden Abschnitten 7.2.4.2, 7.2.4.3 und 7.2.4.4.

7.2.4.1 *Aufbau*

Für die Erhebung der Daten wurde, wie bereits erläutert, der Linux-Kern in Version v5.4 auf der i386-Architektur genutzt. Durchgeführt wurde die Datenerhebung und -analyse auf einem Mehrkernsystem mit einem Intel ® Xeon ® E5-1620 Prozessor. Als Arbeitslast wurde aufgrund der Analysen aus Unterabschnitt 5.1.2 die Testsuite *syscalls*[13] aus dem *Linux Test Project* [H+] genutzt. Aufgrund von Problemen bei der Ausführung in der virtualisierten Umgebung mussten verschiedene Tests der Suite *syscalls* deaktiviert werden. Da diese jedoch nur 0,04 % zu der gesamten Basisblock-Abdeckung beitragen, ist die Einschränkung zu vernachlässigen. Die von *syzkaller* generierten Programme kommen an dieser Stelle jedoch noch nicht zum Einsatz, da es hier ebenso technische Probleme in der virtualisierten Umgebung gibt. Als Auswahlstrategie wurde die Strategie *Bottom Up* mit einem Akzeptanzschwellenwert von $t_{ac} = 0,985$ verwendet. Es kommt eine kontext-sensitive Nachverarbeitung zum Einsatz. Die Zugriffe werden nach der Variante *aggregiert* gezählt.

Tabelle 7.9 gibt eine Übersicht über die Metadaten zur Datenerhebung und -verarbeitung mittels FAIL* für den Linux- und FreeBSD-Kern. Die Kosten für die Instrumentierung spiegeln sich in der deutlichen höheren Laufzeit der Arbeitslast wieder, wenngleich die Laufzeit für Datenerhebung noch den Startvorgang des Kerns beinhaltet.

7.2.4.2 *Überprüfung der Sperren-Regeln*

Im ersten Schritt der LockDoc-Analyse wird die existierende Sperren-Dokumentation im Linux-Kern untersucht [LSBS19]. Hierzu werden 8 zentral dokumentierte Datentypen herangezogen [LSBS19]. Dabei handelt es sich

13 Es wurde das Git-Tag *20190115* aus dem LTP-Repository verwendet.

Datum		Linux	FreeBSD
Laufzeit der Arbeitslast	aufgezeichnet	20,56 Stunden	26,43 Stunden
	real (in einer VM)	10,88 min	20,22 min
#Ereignisse		1,61 Mrd.	1,52 Mrd.
#Sperren-Operationen		433,97 Mio.	803,82 Mio.
#Allokationen		1,45 Mio.	0,19 Mio.
#Freigaben		1,43 Mio.	0,19 Mio.
#Speicherzugriffe	aufgezeichnet	1,17 Mrd.	720,24 Mio.
	nicht gefiltert	1,13 Mrd.	720,24 Mio.
Laufzeit der Nachverarbeitung		15,38 Stunden	14,42 Stunden
Filter und Importieren der Daten		14 Stunden	6 Stunden
Ableiten der Sperren-Regeln		1 Stunden	2 Stunden
Gegenbeispiele bestimmen		21 min	6 Stunden

Tabelle 7.9: Übersicht über die wichtigsten Metadaten zu der Datenerhebung und -verarbeitung in Linux sowie in FreeBSD.

um die Datenstrukturen `struct journal_t`, `struct transaction_t`, `struct inode`, `struct dentry`, `struct journal_head`, `struct super_block`, `struct jbd2_inode` sowie `struct backing_dev_info`[LSBS19]. Alle Datentypen stehen im Zusammenhang mit dem VFS-Subsystem des Linux-Kerns [LSBS19]. Ihre Dokumentation findet sich in den ersten Zeilen von `fs/inode.c`, `fs/dcache.c`, `include/linux/backing-dev-defs.h`, `include/linux/journal-head.h` sowie in `include/linux/dcache.h` (Zeile 89 ff.) [LSBS19]. Die Dokumentation der Datentypen `struct journal_t` und `struct transaction_t` sind besonders hervorzuheben: Ihre Sperren–Dokumentation enthält zu fast jedem Element eine Annotation in der Datei `include/linux/jbd2.h` (in den Zeilen 572 ff. und 761 ff.) [LSBS19].

Tabelle 7.10 stellt eine Zusammenfassung der Ergebnisse dar[14]. In Summe werden 172 Sperren-Regeln (Spalte #R) herangezogen. Die Spalten #Ob und #No geben an, wie viele Regeln während der Aufzeichnung beobachtet bzw. nicht beobachtet wurden. Die Regeln, die nicht beobachtet wurden, sind u.a. auf fehlende Speicherzugriffe zurückzuführen. Hier war die Arbeitslast nicht in der Lage für die Ausführung der relevanten Codepfade zu sorgen. Es kann aber auch Zugriffe gegeben haben, die aber im Rahmen der Nachverarbeitung gefiltert wurden. Für alle beobachteten Sperren-Regeln wird anschließend bestimmt, ob sie zu 100 % durch die Beobachtungen gedeckt

14 Vereinzelte aus dieser Arbeit resultierende Änderungen am Linux-Kern wurden hierbei bereits berücksichtigt.

Datentyp	#R	#No	#Ob	✓(%)	~ (%)	✗(%)
journal_t	38	6	32	53,13	37,50	9,38
transaction_t	36	6	30	80,00	20,00	0,00
inode	20	2	18	33,33	55,55	11,11
dentry	24	4	20	25,00	75,00	0,00
journal_head	26	2	24	58,33	16,67	25,00
super_block	4	0	4	75,00	25,00	0,00
jbd2_inode	12	0	12	91,67	8,33	0,00
backing_dev_info	12	1	11	72,73	27,27	0,00

Tabelle 7.10: Zusammenfassung der Überprüfung der Sperren-Regeln: Jede Zeile zeigt auf, wie viele Sperren-Regeln dokumentiert (#R), wie viele davon beobachtet (#Ob) bzw. nicht beobachtet wurden (#No). Eine Sperren-Regel bezieht sich immer auf ein Tupel aus Datentyp, Element und Zugriffstyp. Die weiteren Spalten geben jeweils den Anteil an Regeln an, die korrekt ($s_r = 1$), mehrdeutig ($0 < s_r < 1$) oder falsch ($s_r = 0$) sind. (Nach einer Vorlage von Lochmann et al. [LSBS19])

werden (Spalte ✓), sie teilweise durch die Beobachtungen gedeckt werden (Spalte ~) oder ob es gar keine Beobachtung der Regel folgt (Spalte ✗). Die Tabelle 7.11 zeigt anhand der Datenstruktur `struct inode`, wie diese Ergebnisse im Detail aussehen.

Es konnte gezeigt werden, dass von den beobachteten Regeln (siehe Spalte *#Ob*) in 53,12 %, 80 %, 33,33 %, 25 %, 58,33 %, 75 %, 72,73 % sowie 91,67 % der Fälle einen relativen Support von 100 % haben ($s_r = 1$). Insgesamt folgt der Programmcode in 58,28 % der Fälle den dokumentierten Regeln. Bei den mehrdeutigen Regeln (Spalte ~) muss im Detail im Austausch mit den Entwicklern geklärt werden, ob die Dokumentation oder der Ansatz versagen.

Lediglich die Datenstrukturen `struct inode`, `struct journal_head` und `struct journal_t` weisen Regeln auf, die gar nicht in den Aufzeichnungen auftauchen. Im Fall der Struktur `struct journal_head` wurde bereits eine Änderung eingereicht, die diesen Mangel behebt – vgl. Unterabschnitt 7.2.3. Bei der Datenstruktur `struct inode` ist allein das Element `i_size` betroffen. Da es sich um einen 64-bit-Datentyp[15] handelt, kommt unter der untersuchten Architektur (i386) ein *Sequential Lock* zur Absicherung zum Einsatz[16]. Da laut Dokumentation eigentlich eine andere Sperre nötig ist und dieser Typ nicht unterstützt wird, schlägt der Abgleich der Sperren-Regeln fehl. Für die beiden Elemente, `j_revoke` und `j_revoke_table`, der Struktur `struct journal_t`, deren Regeln nicht in der Aufzeichnung zu finden sind, zeigte sich, dass die Dokumentation

15 Definiert in `linux-v5.4/include/linux/fs.h`.
16 Vgl. `linux-v5.4/include/linux/fs.h`, Zeile 925 ff.

Element	r/w	Sperren-Regel	s_r	OK?
i_sb_list	w	EO(super_block.s_inode_list_lock)	100 %	✓
i_io_list	w	EO(backing_dev_info.wb.list_lock)	100 %	✓
i_dentry	r	ES(inode.i_lock)	100 %	✓
i_dentry	w	ES(inode.i_lock)	100 %	✓
i_bytes	r	ES(inode.i_lock)	100 %	✓
i_bytes	w	ES(inode.i_lock)	100 %	✓
i_state	w	ES(inode.i_lock)	99,98 %	~
i_blocks	w	ES(inode.i_lock)	99,95 %	~
i_hash	w	inode_hash_lock→ES(inode.i_lock)	94,80 %	~
i_lru	r	ES(inode.i_lock)	50,48 %	~
i_sb_list	r	EO(super_block.s_inode_list_lock)	45,86 %	~
i_lru	w	ES(inode.i_lock)	35,53 %	~
i_io_list	r	EO(backing_dev_info.wb.list_lock)	20,22 %	~
i_state	r	ES(inode.i_lock)	18,53 %	~
i_hash	r	inode_hash_lock→ES(inode.i_lock)	18,08 %	~
i_blocks	r	ES(inode.i_lock)	10,96 %	~
i_size	r	ES(inode.i_lock)	0 %	✗
i_size	w	ES(inode.i_lock)	0 %	✗

Tabelle 7.11: Übersicht über die überprüften Sperren-Regeln für die Datenstruktur `struct inode`: *ES* bezeichnet Sperren, die in die Datenstruktur eingebettet sind, die gerade zugegriffen wird. Die Sperre `inode_hash_lock` bezeichnet hingegen eine globale Sperre. (Nach einer Vorlage von Lochmann et al. [LSBS19])

missverständlich ist: Die Sperre schützt nicht den Zugriff auf die Elemente selbst, vielmehr schützt sie den Speicher auf den sie zeigen[17].

Allgemein können die Kern-Entwickler aus den Ergebnissen aus Tabelle 7.10 zwei Dinge lernen: a) Die Dokumentation ist veraltet und muss daher aktualisiert wird werden [LSBS19]. b) Der Programmcode folgt nicht den dokumentierten Regeln, was in den bereits in Unterabschnitt 2.2.2 diskutierten Folgen resultieren kann [LSBS19]. Im Rahmen dieser Arbeit zeigte sich, dass keine richtige, zentrale Sperren-Dokumentation existiert [LSBS19]. Viel mehr ist das Wissen über die korrekten Sperren-Regeln über die zentralen Entwickler verteilt und zeigt sich erst in dem Review-Prozess von eingereichten Änderungsvorschlägen [LSBS19]. Daher kann nicht abschließend geklärt werden, ob entweder die Dokumentation falsch ist oder im Programmcode die falschen Regeln angewendet werden [LSBS19]. Allerdings gelang es dem Autor dieser Arbeit, dass ein Änderungsvorschlag[18] zur Ak-

17 Vgl. `linux-v5.4/fs/jbd2/revoke.c`.

18 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=50a4952fd67b7f7f551e82ac07c51c1a7a74d474

tualisierung der Dokumentation der Datenstruktur `struct transaction_t` akzeptiert wird. Die Erkenntnisse hierzu stammen aus dieser Arbeit. Wenngleich der LockDoc-Ansatz für das Schreiben des Elements `t_chp_stats` eine andere Hypothese als Gewinner ausgewählt hat, so hatte die Hypothese, die letztlich in die vorgenannte Änderung eingeflossen ist, dennoch einen relativen Support von 100 %. Der Fehler konnte durch manuelle Inspektion seitens des Autors erkannt werden. Ein größerer Akzeptanzschwellenwert hätte ihn vermieden.

Abseits der oben dargestellten Konsequenzen für die Kern-Entwickler ist natürlich auch dieser Ansatz hinsichtlich möglicher Unschärfe zu hinterfragen. Der folgende Unterabschnitt 7.2.4.3 führt hier verschiedene Gründe auf.

Dennoch kann die Statistik aus Tabelle 7.10 eingesetzt werden, um die Qualität der Sperren-Dokumentation zu beurteilen [LSBS19]. Eine manuelle Untersuchung der Dokumentation zeigte, dass die Datentypen `struct journal_t` und `struct transaction_t` deutlich sorgfältiger dokumentiert sind als z. B. die Datenstruktur `struct inode` [LSBS19]. Daher fallen auch die Werte in Tabelle 7.10 deutlich besser aus [LSBS19]. Im Rahmen der Inspektion fielen drei Elemente der Datenstruktur `struct transaction_t` auf, die von einem einfachen `int`-Datentyp zu einem explizit atomar-lesbaren Datentyp (`atomic_t` [Lov10]) umgewandelt wurden [LSBS19]. Allerdings wurde die Dokumentation hierbei nicht angepasst [LSBS19]. Ein Änderungsvorschlag[19] seitens des Autors dieser Arbeit, der dies nachholt, wurde bereits akzeptiert.

7.2.4.3 *Generierung von Sperren-Regeln*

In diesem Abschnitt werden die abgeleiteten Sperren-Regeln für alle beobachteten Datenstrukturen vorgestellt. Die Regeln können genutzt werden, um eine neue, einheitliche und zentrale Sperren-Dokumentation zu erzeugen. Die Tabelle 7.12 zeigt eine Zusammenfassung der abgeleiteten Sperren-Regeln für alle 33 Datentypen. Die Zusammenfassung für die 22 Unterklassen der Datenstruktur `struct inode` ist in Tabelle 7.13 dargestellt. Datentypen mit o Beobachtungen wurden ausgelassen. Die Spalten #M und #Bl zeigen auf, für wie viele Elemente Sperren-Regeln abgeleitet bzw. gefiltert wurden – vgl. Abschnitt 6.2. Die Spalte #Regeln listet die Anzahl an Regeln mit einem hinreichenden Support ($s_r \geq t_{ac}$ mit $t_{ac} = 0,985$). Wohingegen die Spalte #Nl (= *No Lock*) die Teilmenge der Spalte #Regeln nennt, bei der für die zugehörigen Elemente keinerlei Sperren nötig sind. Das Verringern des Akzeptanzschwellenwertes unter 98,5 % würde natürlich mehr Hypothesen, die nicht der Regel „keine Sperren nötig“ entsprechen, akzeptieren [LSBS19]. Insgesamt konnte für 51,29 % der insgesamt 1.585 möglichen Elemente (Summe über Spalte #E - Spalte #Bl) eine Regel für das Lesen bestimmt werden. D.h. es gab mindestens einen ungefilterten Zugriff. Dabei gewann die leere Hypothese in 74,29 % der Fälle. Analog wurde für 28,83 % der Elemente eine Regel für das Schreiben abgeleitet. Der Anteil der leeren Hypothese lag bei 30,42 %. Es kann zweierlei Gründe für einen hohen Anteil an

19 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=32ea275008d8c

Datentyp	#E	#Bl	#Regeln		#Nl	
			r	w	r	w
backing_dev_info	46	4	30	27	17	12
block_device	23	2	19	15	9	2
buffer_head	13	0	11	9	10	6
cdev	6	0	3	6	2	5
dentry	21	2	18	17	14	8
jbd2_inode	7	0	7	6	2	0
journal_head	15	0	14	12	6	0
journal_t	58	11	40	20	25	3
pipe_inode_info	16	1	15	13	9	2
super_block	59	5	49	37	36	11
transaction_t	27	1	21	15	9	0

Tabelle 7.12: Übersicht über die abgeleiteten Sperren-Regeln für 33 Datentypen exklusive der 22 Unterklassen der Datenstruktur `struct inode`. Jede Zeile gibt die Anzahl der Elemente (#E), für die Regeln abgeleitet wurden, sowie die Anzahl der gefilterten Elemente (#Bl) wieder. Anschließend wird die Anzahl aufgeschlüsselt nach dem Zugriffstyp der abgeleiteten Regeln (#Regeln) aufgelistet. Die letzten Spalten geben Summe der Elemente wieder, die keinerlei Sperren benötigen (#Nl). (Nach einer Vorlage von Lochmann et al. [LSBS19])

leeren Hypothesen geben: a) Es sind wirklich keine Sperren für einen Zugriff erforderlich [LSBS19]. b) Die korrekte Hypothese hat einen zu geringen relativen Support [LSBS19]. Hierfür kann es wiederum verschiedene Gründe geben:

- **Geringer absoluter Support:** Zu wenige oder gar keine Speicherzugriffe können in einem geringen absoluten Support resultieren [LSBS19]. Wie bereits in Unterabschnitt 5.1.2 diskutiert wurde, hängt dies mit der Auswahl der Arbeitslast zusammen [LSBS19]. Bestimmte Codepfade werden in diesem Fall gar nicht ausgeführt. Dies kann in der Zukunft mit einer besseren Arbeitslast z. B. mit den Ergebnissen aus Unterabschnitt 7.2.1.2 verbessert werden [LSBS19]. Natürlich kann es auch Elemente geben, die während der Initialisierung einmal beschrieben und im weiteren Verlauf nur noch gelesen werden. In solchen Fällen kann natürlich keine Regel für das Schreiben abgeleitet werden.
- **Geringer relativer Support:** Es gibt verschiedene Gründe für einen geringen relativen Support: a) Es wurden nicht alle Teile des Kerns korrekt instrumentiert [LSBS19]. b) Die initiale Annahme aus Kapitel 5,

Datentyp	#E	#Bl	#Regeln		#Nl	
			r	w	r	w
inode:aio	67	7	24	8	19	3
inode:anon_inodefs	67	7	16	2	10	0
inode:bdev	67	7	29	17	17	2
inode:debugfs	67	7	5	2	1	2
inode:devpts	67	7	4	0	0	0
inode:devtmpfs	67	7	39	26	33	8
inode:dmabuf	67	7	4	0	0	0
inode:ext2	67	7	47	34	38	9
inode:ext3	67	7	43	23	35	3
inode:ext4	67	7	50	38	39	11
inode:hugetlbfs	67	7	28	17	20	10
inode:mqueue	67	7	24	9	17	4
inode:nsfs	67	7	32	12	29	8
inode:overlay	67	7	30	23	26	11
inode:pipefs	67	7	34	7	31	3
inode:proc	67	7	33	10	31	4
inode:rootfs	67	7	39	14	35	2
inode:selinuxfs	67	7	4	0	0	0
inode:sockfs	67	7	24	3	20	0
inode:sysfs	67	7	34	13	28	1
inode:tmpfs	67	7	39	22	36	9
inode:tracefs	67	7	4	0	0	0

Tabelle 7.13: Übersicht über die abgeleiteten Sperren-Regeln für die 22 Unterklassen der Datenstruktur `struct inode`. Jede Zeile gibt die Anzahl der Elemente (#E), für die Regeln abgeleitet wurden, sowie die Anzahl der gefilterten Elemente (#Bl) wieder. Anschließend wird aufgeschlüsselt nach dem Zugriffstyp die Anzahl der abgeleiteten Regeln (#Regeln) aufgelistet. Die letzten Spalten geben Summe der Elemente wieder, die keinerlei Sperren benötigen (#Nl). (Nach einer Vorlage von Lochmann et al. [LSBS19])

> dass das Sperren innerhalb des Kerns meistens korrekt gehandhabt wird, ist nicht korrekt [LSBS19]. c) Der LockDoc-Ansatz stößt hierbei an seine Grenzen [LSBS19]. Leider erfordern alle diese Gründe eine tief gehende, manuelle Untersuchung des Quellcodes und der Daten [LSBS19]. Zusätzlich erfordert es Kommunikation mit den Domänenexperten, den Kern-Entwicklern, um diese Fragen zu klären [LSBS19].

Ein Beispiel für geringen relativen Support beschäftigt sich mit dem Element `inode.i_hash` [LSBS19]. Gemäß der abgeleiteten Regeln ist die globale Sperre `inode_hash_lock` erforderlich. Allerdings sagt die existierende Dokumentation aus, dass die Sperre `inode.i_lock` nötig ist – vgl. Listing 3.1. In der Funktion `__remove_inode_hash()`[20] werden indes beide Sperren geholt. Die genannte Funktion entfernt eine Inode aus einer doppelt-verketteten Liste. Dazu wird das Element `i_hash` in drei Inode-Instanzen beschrieben. Es wird die zu entfernende Inode, deren Sperre gehalten wird, deren Vorgänger sowie der Nachfolger beschrieben. Allerdings wird die Sperre von den beiden letztgenannten nicht gehalten [LSBS19]. Dies führt zu Schreibzugriffen, ohne die zugehörige Sperre `i_lock` zu halten. In der Konsequenz schließt LockDoc daraus, dass lediglich die Sperre `inode_hash_lock` nötig ist [LSBS19]. Fraglich bleibt, ob es sich dabei um einen Programmierfehler handelt oder ob der Ansatz an dieser Stelle versagt [LSBS19]. Dies kann jedoch erst durch Interaktion mit den Kern-Entwicklern gelöst werden [LSBS19].

Ein weiteres Beispiel rund um das Element `inode.i_count` deckt Verbesserungspotential für den LockDoc-Ansatz auf: Das Element `i_count` ist vom Typ `atomic_t` und kann mittels bestimmter Operationen atomar gelesen und geschrieben werden. Es dient als Referenzzähler für die zugehörige Instanz. Erfolgt jedoch eine Transition von $1 \rightarrow 0$ so wird atomar mit Hilfe der Funktion `atomic_dec_and_lock()`[21] unmittelbar die Sperre `i_lock` geholt, um konkurrierende Kontrollflüsse auszusperren. Abseits der Tatsache, dass LockDoc aktuell Zugriffe auf Elemente des Typs `atomic_t` ignoriert (vgl. Abschnitt 6.2), würde es aufgrund des zu geringen relativen Supports für die betreffende Hypothese diese nicht akzeptieren.

Ein weiteres, abschließendes Beispiel legt nochmal dar, in welcher Richtung der LockDoc-Ansatz verbessert werden kann: Im Austausch über die LockDoc-Ergebnisse mit einem anerkannten Kern-Entwickler stellte sich heraus, im Linux-Kern wird grundsätzlich angenommen, dass Variablen vom Typ `int` bzw. `long` ohne das Halten einer Sperre gelesen werden können, wenn die Konsistenz eine untergeordnete Rolle spielt. Nach seiner Auskunft ist diese Regel jedoch nicht dokumentiert. Als Beleg für diese These führt er die Implementierung der Funktion `atomic_read()`[22] an: `#define atomic_read(v)READ_ONCE((v)->counter)`.

Zum Abschluss dieses Abschnitts wird noch einmal auf den Einfluss von t_{ac} eingegangen: Hier wird der Anteil der leeren Hypothese („keine Sperren nötig“) an der Gesamtheit der Hypothesen für einen Datentyp betrach-

20 Siehe `linux-v5.4/fs/inode.c`, Zeilen 510 ff.
21 Siehe `linux-v5.4/fs/inode.c`, Zeilen 1581 ff.
22 Siehe `linux-v5.4/include/asm-generic/atomic.h`, Zeilen 171.

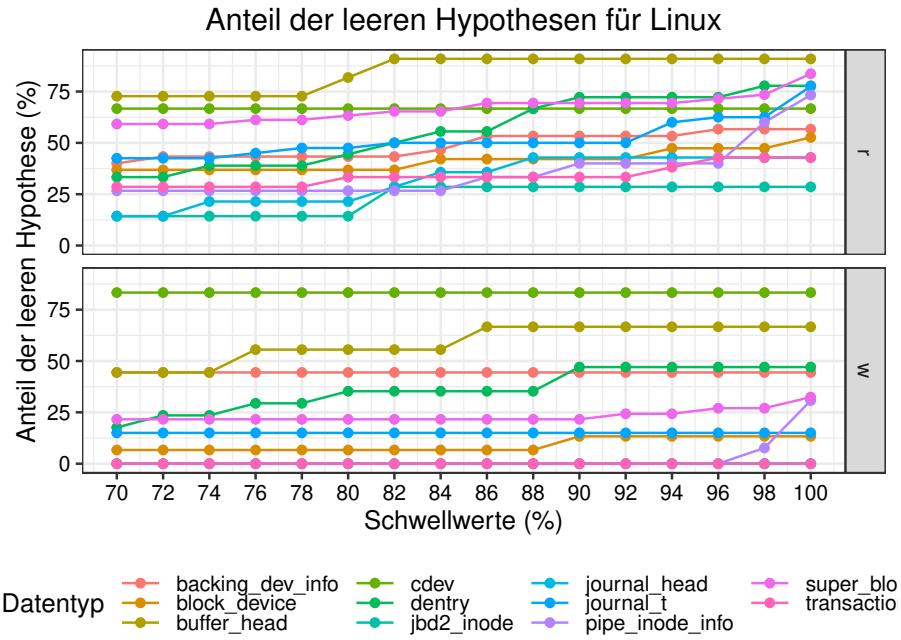


Abbildung 7.4: Aufgetragen wird der Anteil der leeren Hypothese („keine Sperre nötig“) an allen Hypothesen für jeden Datentyp in Linux für verschiedene Werte von t_{ac} – unterschieden nach dem Zugriffstyp. Es wurde die Strategie *Bottom Up* verwendet. (Nach einer Vorlage von Lochmann et al. [LSBS19])

tet [LSBS19]. Wie bereits erläutert, kann LockDoc diese Hypothese als Gewinner auswählen, auch wenn Hypothesen mit Sperren existieren [LSBS19]. Dafür gibt es zwei Gründe: 1) Es sind tatsächlich keine Sperren nötig und die Daten zeigen dies [LSBS19]. 2) Diese Hypothese wird mangels einer alternativen Hypothese mit $t_{ac} \leq s_r$ ausgewählt [LSBS19]. Dementsprechend hängen der Wahl von t_{ac} und der Anteil der Hypothese „keine Sperren nötig“ zusammen [LSBS19]. In Abbildung 7.4 wird der eben genannte Anteil der leeren Hypothese als Gewinner für $0,7 \leq t_{ac} \leq 1$ dargestellt. Die Auswertung wurde für alle Datentypen außer den 22 Unterklassen der Datenstruktur `struct inode` durchgeführt. Für die meisten Datenstrukturen pendelt sich der Anteil bei $t_{ac} = 0,9$ ein. Bei der Hälfte der Elemente des Typs `struct dentry` beispielsweise gewinnt für Schreibzugriffe bei einem Wert von $t_{ac} = 1$ die leere Hypothese. Andersherum bedeutet dies, dass die andere Hälfte der Gewinnerhypothesen einen relativen Support von 100 % aufweist.

Insgesamt ist das Grundniveau in der Darstellung für den schreibenden Zugriff geringer im Vergleich zum lesenden Zugriff. Ein möglicher Grund ist die strengere Einhaltung eines synchronisierten Zugriffs, wenn das Element geschrieben werden soll. Somit gäbe es insgesamt mehr Beobachtungen, die einer bestimmten Sperren-Regel folgen. Dies wiederum verhindert, dass für steigende Akzeptanzschwellenwerte die leere Hypothese gewinnt. Andersherum stützt die größere Dynamik in dem Graph für lesenden Zugriffe die Hypothese über den einfachen, unsynchronisierten Zugriff auf Integer-Datentyp im Linux-Kern: Werden mehr Zugriffe ohne jegliche Sperre oder

mit zufällig gehaltenen Sperren durchgeführt, so erhält keine Hypothese den nötigen relativen Support, um sich dauerhaft bei steigendem Akzeptanzschwellenwert durchzusetzen. Somit gewinnt häufiger die leere Hypothese und erhält insgesamt einen größeren Anteil.

7.2.4.4 *Verstöße gegen Sperren-Regeln*

Für jede abgeleitete Sperren-Regel aus Tabelle 7.13, die einen relativen Support $s_r < 1$ hat, gibt es Zugriffe, die nicht den Regeln folgen [LSBS19]. Dementsprechend kann LockDoc für all diese Fälle die Position, Aufrufhierarchie sowie die tatsächlich gehaltenen Sperren ermitteln – vgl. Unterabschnitt 6.4.3. Tabelle 7.14 gibt eine Übersicht über alle Gegenbeispiele, die gemäß der Sperren-Regeln aus Tabelle 7.12 sowie 7.13 anfallen [LSBS19]. Für jeden Datentyp werden die Gesamtanzahl an Speicherzugriffen, die involvierten Elemente sowie die verschiedenen Kontexte (Funktion plus Aufrufhierarchie) aufgelistet [LSBS19]. Eine Funktion wird entsprechend mehrfach gezählt, wenn es bei jedem Zugriff eine andere Aufrufhierarchie gab [LSBS19]. Passend dazu gibt Tabelle 7.15 mehrere Beispiele, wie genau die Gegenbeispiele aussehen [LSBS19]. Hier dargestellt sind Gegenbeispiele zu der Sperren-Regel für das Lesen von drei verschiedene Elementen [LSBS19]. Wie hier zu sehen ist wird das Element `j_committing_transaction` beispielsweise ohne jegliche Sperre gelesen. Das Element `i_acl` wird z. B. mit drei gehaltenen Sperren gelesen, obwohl die Ergebnisse keine Sperren prognostizieren.

In Linux wurde für eine Datenstruktur ein nicht-abgesicherter, lesender Zugriff auf ein Datenstrukturelement eingebaut. Dieser wurde von LockDoc auch als Gegenbeispiel erkannt. Abseits dieses einen Gegenbeispiels ist es aber offensichtlich, dass es nicht bei allen der in Summe 43.953 Gegenbeispiele um echte Synchronisationsfehler handelt [LSBS19]. Eine manuelle Untersuchung der fünf Datentypen mit einer vier- oder fünfstelligen Anzahl an Gegenbeispielen bestätigte dies. Hierfür gibt es verschiedene Gründe für die falsch-positiven Ergebnisse:

1. Im vorangegangenen Unterabschnitt 7.2.4.3 wurde bereits das Problem der unsynchronisierten Zugriffe auf Elemente mit Wortbreite diskutiert. Werden bei diesen Zugriffen in genügend Fällen zufällig andere Sperren gehalten, so wird es eine Gewinnerhypothese verschieden von der leeren Hypothese geben. Somit werden die verbleibenden Zugriffe als Fehler eingestuft [LSBS19]. Der Vorschlag des Autors dieser Arbeit, ein separates Makro für solche Zugriffe einzuführen, wurde mit Verweis auf das Makro `READ_ONCE()` abgelehnt[23]. Dieses Makro sei zur Kennzeichnung solcher Zugriffe gedacht, allerdings würde es nur für neue Zugriffe verwendet. Das Problem für das maschinelle Überprüfen durch die große Anzahl der nicht-abgesicherten Zugriffe, die nicht markiert sind, werde von den Kern-Entwicklern nur als theoretisch angesehen.

23 https://www.spinics.net/lists/linux-ext4/msg76659.html

Datatyp	Häufigkeit	Elemente	Kontexte
backing_dev_info	5059	1	9
block_device	18	3	8
inode:bdev	3188	5	54
inode:devtmpfs	4	2	4
inode:ext2	5082	7	261
inode:ext4	16256	8	107
inode:overlay	86	3	42
inode:proc	74	2	20
inode:rootfs	32	3	16
inode:sysfs	3	3	3
inode:tmpfs	5	2	3
inode:tracefs	2	2	2
journal_t	13274	6	149
pipe_inode_info	84	3	20
super_block	786	6	29

Tabelle 7.14: Übersicht über die Gegenbeispiele zu den abgeleiteten Sperren-Regeln unter Linux. Pro Zeile wird die Anzahl der falsch-abgesicherten Zugriffe, die dabei betroffenen Elemente sowie die Anzahl der verschiedenen Kontexte aufgelistet. Insgesamt gab es 43.953 Zugriffe aus 727 Kontexten. Datentypen ohne Gegenbeispiele wurden ausgelassen. (Nach einer Vorlage von Lochmann et al. [LSBS19])

2. Wenngleich im Rahmen der Nachverarbeitung von LockDoc bereits Zugriffe im Kontext von Objektinitialisierung und Objektzerstörung gefiltert werden, so gibt es noch weitere Stellen im Linux-Kern, in denen absichtlich Sperren-Regeln aus Performanzgründen ignoriert werden [LSBS19]. Dies kann z. B. der Fall sein, wenn aus dem Kontext des Zugriffs klar ist, dass es keine konkurrierende Zugriff geben kann [LSBS19].

3. Eine punktuelle Inspektion der Daten zeigte, dass die korrekte Hypothese einen relativen Support von 100 % aufwies. Aufgrund der Auswahlstrategie und der Wahl von t_{ac} wird jedoch eine Hypothese mit einem kleineren, relativen Support als Gewinner ausgewählt. Entsprechend sind die zu der Hypothese gehörenden Gegenbeispiele höchstwahrscheinlich falsch-positive Ergebnisse.

Datentyp / Element	Sperren	Codeposition
inode:ext4.i_acl	EO(super_block.s_umount)→ EO(inode:ext4.i_rwsem)→ ES(inode:ext4.i_rwsem)	fs/posix_acl.c, Zeile 68
journal_t. j_committing_transaction	keine Sperren	fs/jbd2/commit.c, Zeile 394
super_block.dq_op	ES(super_block.s_umount)	fs/quota/dquot.c, Zeile 901

Tabelle 7.15: Beispiele für lesende Gegenbeispiele zu den abgeleiteten Sperren-Regeln: Für jeden falsch-synchronisierten Zugriff wird der Kontext (Funktion und Aufrufhierarchie) sowie die tatsächlich gehaltenen Sperren angegeben. *ES* steht für *embedded same* und bezeichnet Sperren, die in die gerade zugegriffene Datenstruktur eingebettet sind. Analog dazu steht *EO* für *embedded other*. (Nach einer Vorlage von Lochmann et al. [LSBS19])

4. Der LockDoc-Ansatz bildet noch nicht alle Mechanismen zur Synchronisation ab. Oder die aktuelle Instrumentierung spart Teile des Linux-Kerns aus, die für die korrekte Bewertung nötig wären.

Ein Beispiel für den ersten Grund ergab sich im Rahmen einer Diskussion mit den Kern-Entwicklern[24]: Das Element `j_running_transaction` der Struktur `struct journal_t` enthält einen Zeiger. Das Lesen des Zeigers ist problemlos ohne Sperre möglich. Soll der Zeiger jedoch dereferenziert werden, so muss eine Sperre geholt werden. Passiert der erste Fall zu häufig, schlussfolgert LockDoc, es seien keine Sperren nötig. Die Diskussion um einen Fehlerreport[25], der eine Wettlaufsituation beim Zugriff auf das Element `b_transaction` der Datenstruktur `struct journal_head` meldet, ist als weiteren Beleg für den ersten Grund zu nennen. Wie durch einen zuständigen Kern-Entwickler aufgeklärt[26] wurde (und ebenfalls im Quellcode vermerkt ist), handelt es sich dabei um absichtlich unsichere Zugriffe. Als Folge aus der Diskussion wurden zwei Änderungen[27] von eben genanntem Entwickler eingereicht, die die erwähnten Zugriffe mit einem Makro versehen. Diese Zugriffe werden so für das zuständige Werkzeug als falsch-positiv gekennzeichnet.

Zu 2. lässt sich ein Beispiel nennen, das sich ebenfalls aus dieser Diskussion ergab. Der sogenannte *jbd2*-Thread benötigt keinerlei Sperren, um bestimmte Elemente zu lesen bzw. zu schreiben. Auch hier führt eine Häufung dieser Zugriffe zu einer „falschen“ Gewinnerhypothese. Ein weiteres Beispiel ist die Funktion `evict()`[28]. Ihr Name sowie der Kontext, in dem sie verwendet wird, suggerieren, dass es sich um die Freigabe einer Inode handelt. Dementsprechend sollten Zugriffe innerhalb dieser Funktion ignoriert

24 https://www.spinics.net/lists/linux-ext4/msg76657.html
25 https://lkml.org/lkml/2021/4/4/42
26 https://lkml.org/lkml/2021/4/6/537
27 https://www.spinics.net/lists/linux-ext4/msg77226.html
https://www.spinics.net/lists/linux-ext4/msg77225.html
28 Implementiert in `linux-v5.4/fs/inode.c`, Zeile 568 ff.

werden. Allerdings werden innerhalb der Funktion noch Sperren eingesetzt, so dass nicht ganz klar ist, ob eine Filterung angebracht ist oder nicht. Vielleicht müssen auch nur die Zugriffe für eine Teilmenge der Elemente gefiltert werden. Dies ist in der Zukunft im Dialog mit den Entwicklern zu klären.

Bisher noch nicht in LockDoc abgebildet sind Referenzzähler. Sie verhindern, dass der zugrundeliegende Speicher freigegeben wird, während es noch Zugriffe geben kann. Das Element `i_count`[29] der Datenstruktur `struct inode` ist ein Beispiel dafür. Ist nun für den Zugriff auf ein Element lediglich das Inkrementieren des Referenzzählers erforderlich, so wird LockDoc alle zufällig gehaltenen Sperren auswerten. Der LockDoc-Ansatz wird in diesem Fall versagen, da eine Hypothese mit zufällig gehaltenen Sperren gewinnen wird – oder eben die leere Hypothese.

Zusammenfassend bedeuten diese Argumente, dass auch hier Hilfe von den Domänexperten erforderlich ist [LSBS19]. Da es keine belastbaren Daten zu der Dokumentation gibt und es eine unscharfe Definition gibt, was eine korrekte Synchronisation ist, kann eine falsch-positiv Rate schwer bestimmt werden [LSBS19].

Dennoch gelang es einen echten Synchronisationsfehler[30] zu finden und durch den Autor dieser Arbeit auch zu beheben. Die entsprechende Änderung[31] ist mittlerweile Teil des offiziellen Linux-Kerns.

7.2.5 *Zusammenfassung*

In diesem Abschnitt wurde die Anpassung und Evaluation des LockDoc-Ansatzes auf den Linux-Kern vorgestellt. Alle drei Auswertungen (Unterabschnitt 7.2.4.2, 7.2.4.3 und 7.2.4.4) zeigen vielversprechende Ergebnisse, um die Sperren-Dokumentation in einem derart großen und verteilten Software-Projekt wie dem Linux-Kern langfristig zu verbessern. Insbesondere die fünf akzeptierten Änderungen sowie die eine für gut befundene Änderung sprechen für den Ansatz. Besonders hervorzuheben ist, dass mit vier Änderungen direkt Ergebnisse von LockDoc die Sperren-Dokumentation verbessert haben.

Es wurde aber auch deutlich, dass initial Arbeit geleistet werden muss, um ein Betriebssystem zu unterstützen – vgl. Abschnitt 7.2.2. Hierzu zählt insbesondere die Verfeinerung der Filterlisten. Darüber hinaus zeigte der Ansatz Schwächen bei der Handhabung von Implementierungsdetails in Linux – vgl. Unterabschnitt 7.2.4.3. An dieser Stelle muss noch ein engerer Austausch mit den Linux-Entwicklern erfolgen, um die Abläufe besser zu verstehen. Nur so kann der Ansatz verbessert werden und letztlich belastbarere Aussagen zu dem Synchronisationsverhalten getroffen werden.

---

29 Definiert in `linux-v5.4/include/linux/fs.h`, Zeile 7058.

30 https://lkml.org/lkml/2018/12/7/532
https://lkml.org/lkml/2018/12/14/277

31 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=f69e749a49353d96af1a293f56b5b56de59c668a

7.3 FREEBSD

In diesem Abschnitt wird die Anpassung und Evaluation auf einen weiteren Betriebssystemkern, den von FreeBSD, vorgestellt. Analog zu Abschnitt 7.2 werden auch hier zunächst die erforderlichen Anpassungen an dem FreeBSD-Kern in Abschnitt 7.3.1 erläutert. Darauf folgt die Evaluation der Auswahlstrategien in Unterabschnitt 7.3.2. Ebenso wird der Evaluationsaufbau in Unterabschnitt 7.3.3.1 dargestellt. Darauf folgt in Unterabschnitt 7.3.3.2, 7.3.3.3 und 7.3.3.4 die Präsentation der Evaluationsergebnisse. Den Abschluss bildet auch hier eine kurze Zusammenfassung der Erkenntnisse.

7.3.1 *Anpassungen an das Betriebssystem*

Für die Analyse von FreeBSD wurde der Kern in Version 13.0[32] (i386) verwendet. Wenngleich sich die Version 13.0 von FreeBSD zum derzeitigen Zeitpunkt noch in der Entwicklung befindet, wurde sie dennoch ausgewählt, da sie Unterstützung für das Werkzeug *KCOV* bietet.

Mit dem sog. *Witness*-System [MNNW14] verfügt der FreeBSD-Kern über einen zu Linux verwandten Mechanismus zur Vermeidung von Verklemmungen – vgl. Abschnitt 4.2. Die interne Modellierung der Sperren entspricht hier im Wesentlichem dem Modell aus Abschnitt 5.2 [MNNW14]. Daher wurde anstatt der Programmierschnittstelle der Sperren das *Witness*-System selbst geeignet instrumentiert, um die nötigen Daten zu erheben. Somit werden automatisch alle Sperren, die im *Witness*-System abgebildet sind, automatisch unterstützt. Natürlich wird auch hier die synthetische Sperre *hardirq* unterstützt, die das An- und Abschalten der Unterbrechungen repräsentiert. Tabelle 7.16 gibt einen Überblick über unterstützten Synchronisationsmechanismen und deren Abbildung auf das Sperren-Modell aus Abschnitt 5.2. Damit auf diesem Wege auch die Analyse der Datenstruktur `struct buf` gelingt, musste eine Anpassung an dem *Witness*-System vorgenommen werden: Das *Witness*-System [MNNW14] von FreeBSD überwacht u.a., dass derselbe Kontext einen Mutex holt und ihn wieder freigibt. Repräsentiert jedoch eine Instanz der Struktur `struct buf` einen asynchronen E/A-Vorgang, so erfolgt das Holen und Freigeben der Sperre `b_lock` in unterschiedlichen Kontexten.

Die Freigabe erfolgt in einem separaten Task des Kerns, der die abgeschlossenen E/A-Operation behandelt. Damit es hier nicht zu einer Warnung kommt, wechselt die Sperre, kurz bevor die E/A-Operation abgesetzt wird, den Besitzer[33]. Der Kern wird der neue Besitzer. Für das Witness-System stellt sich diese Operation wie eine Freigabe der Sperre dar und die eigentliche Freigabe wird für Sperren, die dem Kern gehören, nicht mehr dem Witness-System gemeldet. Ein FreeBSD-Entwickler bestätigte dieses Verhalten auf Nachfrage[34]. Da für LockDoc jedoch das Witness-System instru-

32 Basierend auf Git-Commit *2134e85bc1b02389b462c2c9995af98caobf7213*.
33 Siehe `sys/sys/buf.h`, Zeile 394 ff.
34 http://docs.FreeBSD.org/cgi/mid.cgi?YEy3JET6Lx7BkJP3

Modell	Sperren-Typen
Leser-Sperre	-
Schreiber-Sperre	`sleep mutex`, `spin mutex`, `hardirq`
Leser-Schreiber-Sperre	`sx`, `rw`, `rm`, `sleepable rm`, `lockmgr`
Nicht abgebildet	Barrieren, atomare Operationen

Tabelle 7.16: Eine Übersicht über die von LockDoc unterstützten Synchronisationsmechanismen in FreeBSD. Jeder Mechanismus wird einem Teil des Modells aus Abbildung 5.1 zugeordnet.

mentiert wurde, stellt es sich für LockDoc so dar, als ob in dem zweiten Kontext keinerlei Sperren gehalten werden. Die Benachrichtigung des *Witness*-Systems wurde nun so angepasst, dass eine Aufzeichnung der Sperren-Operation erst beim wirklichen Freigeben der Sperre stattfindet. Bei dem Wechsel des Besitzers findet nun keinerlei Aufzeichnung durch LockDoc statt.

Für eine bessere Vergleichbarkeit wird im FreeBSD-Kern ebenfalls das VFS-Subsystem untersucht. Hier werden die folgenden 11 Datentypen unterstützt: `struct vnode`, `struct mount`, `struct bufobj`, `struct buf`, `struct bufv`, `struct bufqueue`, `struct bufdomain`, `struct pipepair`, `struct pipe`, `struct fifoinfo`, `struct nc_vnodelocks`. Die Instrumentierung zur Aufzeichnung der Allokation und Freigabe von Objekten beschränkte sich in FreeBSD ebenfalls auf die dafür vorgesehenen Kern-internen Schnittstellen, wie z. B. `getnewvnode()`. Das Ausrollen von Union-Strukturen war vereinzelt ebenfalls nötig. Der Prozess der Instrumentierung hinterlässt subjektiv den Eindruck, dass die Sperren-Regeln im FreeBSD-Kern strikter befolgt werden und es anders als im Linux-Kern weniger Ausnahmen gibt – vgl. Unterabschnitt 7.2.4.3. Gefiltert werden müssen für FreeBSD 76 Funktionen und 37 Elemente. Zu den gefilterten Funktion zählen auch Funktionen zum atomaren Zugriff auf Speicher, wie z. B. `atomic_substract_int`. Die Liste an gefilterten Elementen enthält viele Elemente der Datenstruktur `struct bufdomain`, da diese als konstant bzw. atomar gekennzeichnet sind[35]. In den Programmcode wurden zu Testzwecken drei einzelne, nicht-abgesicherte, lesende Zugriffe für drei Datenstrukturen eingebaut. Alle lassen sich in den Gegenbeispielen wiederfinden.

7.3.2 *Auswahlstrategien*

Dieser Abschnitt präsentiert analog zu Unterabschnitt 7.2.3 die Evaluation der Auswahlstrategien für den FreeBSD-Kern. Der Aufbau der Evaluation sowie die zu erwarteten Ergebnisse wurden bereits in Unterabschnitt 7.2.3 zusammen mit den Ergebnissen für Linux erläutert. Abbildung 7.5 zeigt die Ergebnisse für FreeBSD. Auch hier bestätigt sich die Tendenz, dass Ergebnisse basierend auf der Zählung *aggregiert* besser ausfallen. Die kontext-sensitive

http://docs.FreeBSD.org/cgi/mid.cgi?YEwxoWdSuJpzhONL

35 Vgl. `freebsd/sys/sys/kern/vfs_bio.c`, Zeile 117 ff.

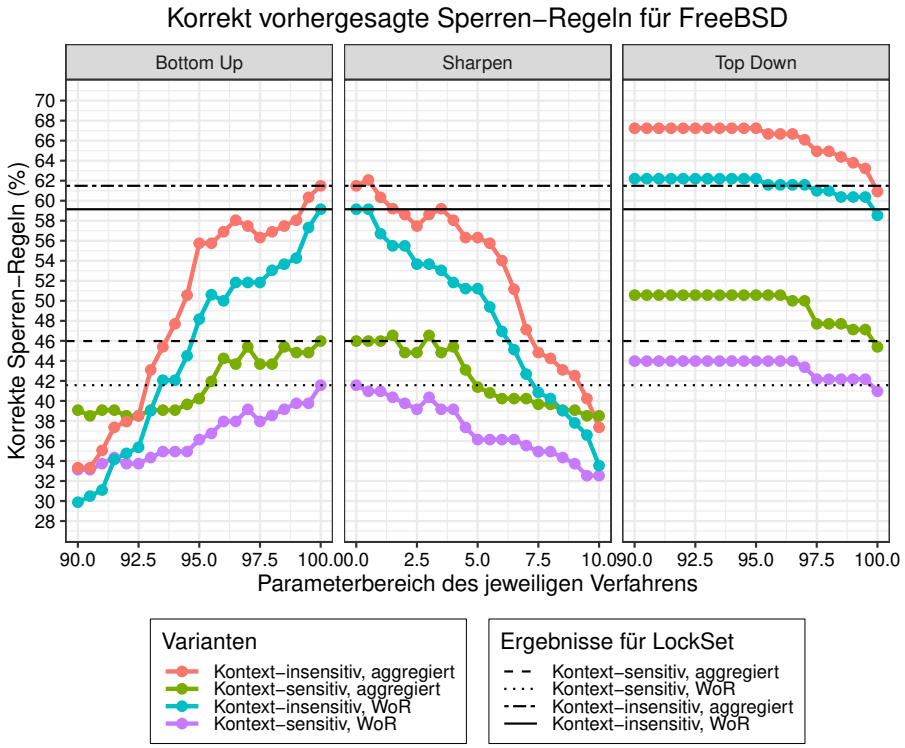


Abbildung 7.5: Auswirkungen der Auswahlstrategien und deren Parameter auf den Anteil der korrekte vorhergesagten Sperren-Regeln für den FreeBSD-Kern: Die y-Achse stellt den Anteil der korrekt vorhergesagten Sperren-Regeln f_h dar. Die x-Achse bildet die untersuchten Parameterwerte ab. Für die Strategien *Top Down* und *Bottom Up* werden Werte im Bereich von $0,9 \leq t_{ac} \leq 1$ gewählt. Für die Strategie *Sharpen* hingegen gilt $0,0 \leq t_s \leq 0,1$. Zum Vergleich stellen die verschiedenen, schwarzen Linien die Ergebnisse der *LockSet*-Strategie je nach Eingabedaten dar.

Nachverarbeitung fällt jedoch für FreeBSD deutlich hinter der kontext-insensitiven Nachverarbeitung zurück. Dies wird im weiteren Verlauf dieses Abschnitts noch untersucht. Die vormals gemachten Annahmen 1. und 2. treffen sowohl für die Strategie *Bottom Up* als auch für die Strategie *Sharpen* zu. Die dritte Annahme ist hier nicht zu beobachten, da die Ergebnisse für f_h monoton ansteigen. Eine mögliche Erklärung hierfür ist, dass es weniger Ausnahmen in Form von unsynchronisierten Zugriffen gibt. Dadurch würde eine korrekte Hypothese einen relativen Support nahe der 100 % haben. Diese würde, je nach Abstand zu anderen Hypothesen, erst für einen restriktiven Parameterwert ausgewählt. Ihr Maximum erreichen die Strategien jeweils für einen restriktiven Parameterwert. Sie liegen damit gleich auf mit den jeweiligen Ergebnissen der *LockSet*-Strategie. Die Strategie *Sharpen* erreicht das Maximum von 62,07 % bereits bei einem Wert für t_s von 0,5 % für die Variante kontext-insensitiv+*aggregiert* (rote Kurve).

Einzig die Ergebnisse der Strategie *Top Down* verlaufen analog zu den Ergebnissen für Linux – vgl. Abbildung 7.3 – und entsprechen auch der dritten Annahme. Auch für FreeBSD zeigt die Strategie *Top Down* bessere Ergebnis-

se als die anderen beiden Strategien. Die Gründe hierfür sollen exemplarisch anhand der Resultate der Variante kontext-insensitiv+*aggregiert* für die Strategien *Sharpen* mit $t_s = 0,005$ und *Top Down* mit $t_{ac} = 0,9$ untersucht werden: Insgesamt werden für acht zusätzliche Tupel die Sperren-Regeln korrekt vorhergesagt. In sechs der Fälle ist die Parameterwahl mit $t_s = 0,005$ zu restriktiv, so dass jedes Mal die leere Hypothese gewinnt. Die korrekte Hypothese hat jeweils einen relativen Support von 95,5-99,1 %. In den anderen beiden Fällen haben die richtigen Hypothesen einen relativen Support von 100 %. Es wird allerdings jeweils eine andere Hypothese ausgewählt, die zu nah an ebendieser liegt. Der Parameter ist mit $t_s = 0,005$ nicht restriktiv genug gewählt. Da die Strategie *Top Down* die Liste der Hypothesen hingegen von oben betrachtet, wird hier die richtige Hypothese ausgewählt. Eine exemplarische Untersuchung des etwas geringeren relativen Supports der korrekten Hypothese für das Schreiben des Elements `b_vflags` der Datenstruktur `struct buf` förderte einen möglichen Programmierfehler zu Tage: Die laut Dokumentation korrekte Hypothese hatte einen zu geringen relativen Support von $s_r = 0,973$, als dass sie für die Wahl von $t_s = 0,005$ ausgewählt werden würde. Statt der leeren Gewinnerhypothese wurde daher in diesem Fall die dokumentierte Sperren-Regel für die Anfrage an die Datenbank zur Bestimmung der Gegenbeispiele genutzt. Dabei stellte sich heraus, dass alle Zugriffe, die nicht der vorgeschriebenen Regel folgen, von genau einer Zeile Programmcode stammen. Die mehreren tausend Ausführungen ohne die korrekte Sperre drückten den relativen Support der richtigen Hypothese so weit, dass er außerhalb der Marge lag. Auf Nachfrage[36] bestätigten die Entwickler[37], dass es mutmaßlich um eine valide Wettlaufsituationen beim Zugriff auf das Element `b_vflags` handelt. Die seitens der Entwickler vorgeschlagene Änderungen wurde in den Quellcode übernommen[38].

Auffällig ist die Spitze bei 3,5 % für die Strategie *Sharpen* in der Variante kontext-insensitiv+*aggregiert*. Eine Untersuchung der Ergebnisse zeigte, dass zunächst bei dem Sprung von 4,5 % auf 3,5 % die Gewinnerhypothese von fünf Tupeln (Datentyp, Element, Zugriffstyp) auf die richtige Hypothese wechselt. Hier gestattete der Wert von $t_s = 0,045$ einen zu großen Schlupf, so dass Hypothesen mit zufällig gehaltenen Sperren ausgewählt wurden. Bei dem Sprung 3,5 % auf 2,5 % wird für zwei der fünf Tupel nun die falsche Sperren-Regel ausgewählt, da der Parameter $t_s = 0,025$ zu restriktiv für den relativen Support der richtigen Hypothesen ist. Aus dem gleichen Grund fällt ein weiteres Tupel raus, für das jedoch vorher ($t_s > 0,045$) schon die korrekte Regel ausgewählt wurde.

Die Spitze bei 3,0 % für Strategie *Sharpen* in der Variante kontext-sensitiv-+*aggregiert* wurde ebenfalls untersucht. Hierbei zeigte sich, dass zunächst bei dem Sprung von 3,5 % auf 3,0 % die Gewinnerhypothese von drei Tupeln (Datentyp, Element, Zugriffstyp) von einer zwei-Sperren-Hypothese auf die richtige Hypothese mit jeweils einem relativen Support von 100 % wechselt. Hier gestattete der Wert von $t_s = 0,035$ einen zu großen Schlupf, so dass

36 http://docs.FreeBSD.org/cgi/mid.cgi?792c8a3d-8ea6-073f-3fda-b3eb793ef2b9
37 http://docs.FreeBSD.org/cgi/mid.cgi?YHVxfMrU9lmw3sG9
38 https://github.com/freebsd/freebsd-src/commit/e3d675958539eee899d42438f5b46a26f3c64902

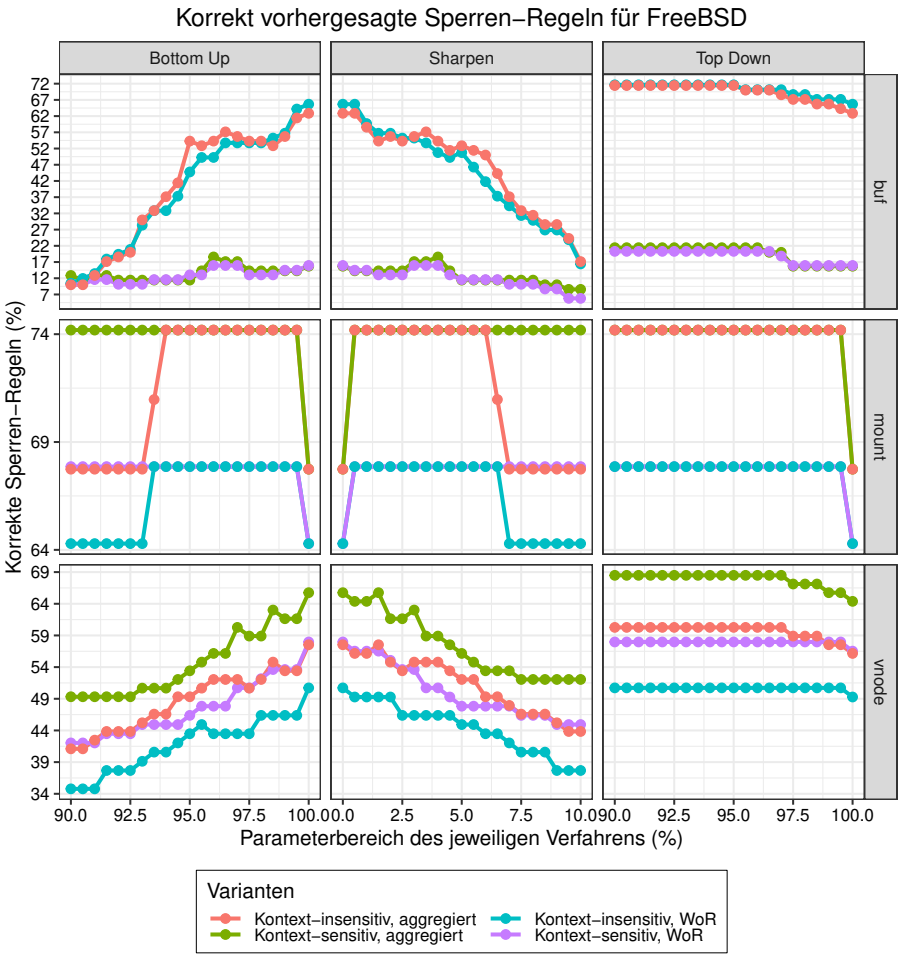


Abbildung 7.6: Auswirkungen der Auswahlstrategien und deren Parameter auf den Anteil der korrekte vorhergesagten Sperren-Regeln für den FreeBSD-Kern: Die y-Achse stellt den Anteil der korrekt vorhergesagten Sperren-Regeln f_h dar. Die y-Achse bildet die untersuchten Parameter werte ab. Für die Strategien *Top Down* und *Bottom Up* werden Werte im Bereich von $0,9 \leq t_{ac} \leq 1$ gewählt. Für die Strategie *Sharpen* hingegen gilt $0,0 \leq t_s \leq 0,1$. In dieser Darstellung werden die Ergebnisse zusätzlich noch nach der betreffenden Datenstruktur aufgeschlüsselt.

Hypothesen mit zufällig gehaltenen Sperren ausgewählt wurden. Ähnliches geschieht bei dem Sprung 3,0 % auf 2,5 %: Für ein Tupel wechselt die Gewinnerhypothese von einer „zufälligen“ Hypothese zu der korrekten Hypothese mit 100 % relativem Support. Allerdings wird bei vier Tupeln die leere Hypothese anstatt der korrekten Hypothese als Gewinner ausgewählt, da der relativen Support der richtigen Hypothesen, jeweils ~97 %, nun außerhalb der erlaubten Marge liegt. Unter dem Strich wird damit der vorher erzielte Gewinn für f_h egalisiert.

Eine weitere deutliche Auffälligkeit ist der große Abstand zwischen den Ergebnissen für die kontext-sensitive und kontext-insensitive Nachbereitung (grüne/violette vs. rote/blaue Kurve). Die erstgenannte Variante bewegt sich im Bereich von 42-50 %. Wohingegen mit der kontext-insensitive Nachver-

arbeitung Ergebnisse von 59-66 % erzielt werden. Weshalb dieser deutliche Unterschied zustande kommt, kann mit Hilfe der Aufschlüsselung nach Datentypen in Abbildung 7.6 erläutert werden: Für die Datenstrukturen `struct vnode` und `struct mount` liefert die kontext-sensitive Nachverarbeitung ähnliche oder bessere Ergebnisse als die kontext-insensitive Nachverarbeitung. Lediglich für die Datenstruktur `struct buf` fallen die Ergebnisse für die kontext-sensitive Nachverarbeitung deutlich schlechter aus. Sie bewegen sich im Bereich von 12-17 %. Die Resultate für die kontext-insensitive Verarbeitung hingegen steigen bis auf knapp 67 % an. Sie sind somit für den deutlichen Unterschied in Abbildung 7.5 verantwortlich. Dies lässt sich mit der Besonderheit bezüglich der Freigabe der Sperre `b_lock` aus Abschnitt 7.3.1 begründen: Die Freigabe der Sperre `b_lock` wird erst aufgezeichnet, wenn sie wirklich stattfindet. Dies geschieht in einem anderen Kontext als das Holen der Sperre. Dadurch gilt die Sperre immer noch als gehalten. Da es bei der kontext-insensitiven Nachverearbeitung nur einen Transaktionsstapel gibt, spielen die unterschiedlichen Kontexte keine Rolle. Alle Zugriffe aus dem zweiten Task, der den E/A-Vorgang abschließt und die Sperre freigibt, lassen sich mit der Sperre in Verbindung bringen. Bei der kontext-sensitiven Nachverarbeitung liegt die Transaktion auf dem Stapel des ersten Tasks, der den E/A-Vorgang begonnen hat. Daher lassen sich die Zugriffe nicht zuordnen. Für LockDoc finden sie entweder ohne jegliche Sperren (leerer Transaktionsstapel) oder mit zufällig, anderen gehaltenen Sperren statt.

Allerdings ist die kontext-insensitive Nachverarbeitung auch mit Kosten verbunden: Wie bereits in Abschnitt 6.2 erläutert wurde, kommt bei der kontext-insensitiven Verarbeitung lediglich ein Transaktionsstapel zum Einsatz. Hierdurch werden synthetische, lange Hypothesen generiert, aus denen wiederum alle Teilmengen erzeugt werden. In Summe führt dies zu einer wesentlich längeren Liste an Hypothesen, die keinen Informationsgewinn verspricht. Eine Liste mit allen in dem Datensatz beobachteten Hypothesen belegt für die kontext-insensitive Nachverarbeitung 401,54 MiB. Wird der Kontext jedoch berücksichtigt, werden für die gleiche Liste lediglich 2,05 MiB benötigt. Dies erleichtert eine Weiterverarbeitung der Daten.

Die Tabelle 7.17 fasst noch einmal quantitativ für die Strategie *Sharpen* mit $t_s = 0,005$ zusammen, was bisher exemplarisch anhand von Einzelfällen erläutert wurde: Sie listet die fünf häufigsten Gründe, weshalb insgesamt nur 62,07 % der Sperren-Regeln korrekt vorhergesagt werden konnte. In 36 der 66 untersuchten Tupel ist der relative Support der korrekten Hypothese zu gering. Hier muss im Detail – ggf. in Zusammenarbeit mit den Entwicklern – in zukünftigen Arbeiten nochmal überprüft werden, weshalb manche Speicherzugriffe nicht der dokumentierten Sperren-Regel folgen. Eine manuelle Durchsicht der Ergebnisse des Datentyps `struct buf` für $t_s = 0,005$ ergab für einen Großteil der Elemente einen geringen relativen Support für die korrekte Hypothese. Eine Stichprobe für die Elemente `b_qindex` und `b_subqueue` zeigte, dass es in beiden Fällen nicht die eine korrekte Hypothese gibt. Die Dokumentation spricht an der Stelle von „Protected by the buf

Fehlertyp	Häufigkeit
fehlerhafte Dokumentation	1
Strategie versagt	9
zu viele Sperren	20
zu wenig rel. Support	36
sonstige	0

Tabelle 7.17: Aufschlüsselung der Gründe sowie deren Häufigkeit für falsch vorhergesagte Sperren-Regeln für die Strategie *Sharpen* für $t_s = 0{,}005$. Es wurden die falsch vorhergesagten Sperren-Regeln für 66 Tupel aus Datenstruktur, Element und Zugriffstyp ausgewertet.

queue lock“[39]. Bei der angesprochenen Sperre handelt es sich aber nicht um genau eine Sperre. Vielmehr gibt es verschiedene Listen des Typs `struct bufqueue`, die in der Datenstruktur `struct bufdomain`[40] eingebettet sind. Jede Instanz der Struktur `struct bufqueue` beinhaltet seine eigene Sperre. Da jede Liste ein eigenes Element darstellt, wird jede Sperre auch einzeln in LockDoc abgebildet. Somit wird der absolute Support auf die Sperren `bd_subq.bq_lock` und `bd_dirtyq.bq_lock` verteilt. Folglich ist es nicht möglich, dass eine einzelne Hypothese genug relativen Support erhält, um zu gewinnen. Der Fall einer fehlerhaften Dokumentation konnte mit Hilfe mit der FreeBSD-Entwicklern geklärt werden und führte zu einer Änderung an der existierenden Sperren-Dokumentation: Er betrifft das Lesen des Elements `mnt_uppers.tqh_first`, eine verkettete Liste, der Struktur `struct mount`. Nach der Inspektion des Programmcodes und der Rücksprache[41] mit den Entwicklern stellte sich heraus, dass die Sperren-Dokumentation nicht mehr aktuell ist. In Folge der Nachfrage wurde die Dokumentation durch die Entwickler korrigiert[42]. Die korrigierte Regel befindet sich mit genau einer Beobachtung in der Liste der Hypothesen. Alle anderen beobachteten Zugriffe werden mit anderen Sperren durchgeführt. Eine Inspektion des Programmcodes in Verbindung mit dem Datensatz ergab, dass fast alle Zugriffe auf eine Stelle zurückzuführen sind[43]. Hierbei handelt es sich um eine nicht abgesicherte Überprüfung, ob die betreffende Liste leer ist. Im Umkehrschluss bedeutet dies, dass alle beobachteten Sperren-Kombinationen mutmaßlich zufällig zustande kamen. Die Arbeitslast deckt vermutlich nicht die richtigen Teile des Programmcodes ab. Abschließend ist auch für FreeBSD festzuhalten, dass die beiden Heuristiken *Bottom Up* und *Sharpen* nicht die Resultate der Strategie *Top Down* erreichen. Aufgrund der konzeptionellen Schwäche der Strategie *Top Down* wird von dieser jedoch abgesehen. Da die Strategie *Sharpen* minimal bessere Ergebnisse erzielt, wird sie für die weitere Analyse in Abschnitt 7.3 eingesetzt. Der

39 Siehe `sys/sys/buf.h`, Zeile 99 ff.
40 Siehe `sys/kern/vfs_bio.c`, Zeile 107 ff.
41 http://docs.FreeBSD.org/cgi/mid.cgi?YG9+hGpGuF4PUR3s
42 https://github.com/freebsd/freebsd-src/commit/5af1131de7fc18c795ed28e69d9393f78875d3e5
43 Siehe `sys/kern/vfs_subr.c`, Zeile 3811.

Schwellenwert wird auf 0,5 % gesetzt. Ferner wird aufgrund der besseren Ergebnisse die kontext-insensitive Nachverarbeitung gewählt.

7.3.3 *LockDoc-Analyse*

In diesem Abschnitt wird der Aufbau zur Datenerhebung unter FreeBSD, Unterabschnitt 7.3.3.1, erklärt. Diesem folgt eine Auswertung der verarbeiteten Daten nach den in Abschnitt 6.4 vorgestellten Kategorien in den darauf folgenden Abschnitten 7.3.3.2, 7.3.3.3 und 7.3.3.4.

7.3.3.1 *Aufbau*

Die Erhebung und Verarbeitung der Daten in FreeBSD findet auf demselben System, wie bereits in Unterabschnitt 7.2.4.1 beschrieben, statt. Unter FreeBSD kommt allerdings die Testsuite *fs*[44] aus dem *Linux Test Project* [H+] zum Einsatz. Sie läuft mit Hilfe der Linux-Kompatibilitätsschicht in FreeBSD [Pro20]. Da die Linux-Kompatibilitätsschicht lediglich ältere Versionen des Linux-Kerns unterstützt, kam es zu Problemen mit der Testsuite *syscalls*, so dass auf die *fs*-Suite ausgewichen wurde. Als Auswahlstrategie wurde die Strategie *Sharpen* mit einem Schwellwert von $t_s = 0,005$ verwendet. Wie bereits in Unterabschnitt 7.3.2 deutlich wurde, kommt für FreeBSD die kontext-insensitive Nachverarbeitung zum Einsatz. Die Zugriffe werden nach der Variante *aggregiert* gezählt.

Tabelle 7.9 gibt ebenfalls eine Übersicht über die Eckdaten zur Datenerhebung und -verarbeitung für den FreeBSD-Kern. Auch fallen die Kosten für die Instrumentierung durch die deutliche höhere Laufzeit der Arbeitslast auf. Während der Ausführung der Arbeitslast wurden folgende Sperren-Typen beobachtet und dementsprechend aufgezeichnet: `hardirq`, `lockmgr`, `rm`, `rw`, `sleepable rm`, `sleep mutex`, `spin mutex` sowie `sx`.

7.3.3.2 *Überprüfung der Sperren-Regeln*

Analog zu der Auswertung für den Linux-Kern werden in Tabelle 7.19 die Ergebnisse zur Überprüfung der Sperren-Dokumentation im FreeBSD-Kern dargestellt. Betrachtet wird die Sperren-Dokumentation für die Datentypen `struct vnode`, `struct mount` sowie `struct buf`. Die Dokumentation von zwei Datentypen wurde bereits auszugsweise in Listing 3.4 bzw. Listing 3.5 in Abschnitt 3.2 diskutiert. Die Dokumentation findet sich in `sys/sys/vnode.h` (Zeile 98 ff.), `sys/sys/buf.h` (Zeile 99 ff.) sowie in `sys/sys/mount.h` (Zeile 189 ff.). Die zu überprüfende Dokumentation enthält zusätzlich noch den Datentyp `struct bufobj` (`sys/sys/bufobj.h` Zeile 99 ff.). Da die Elemente dieses Typs jedoch in die Struktur `struct vnode` eingebettet sind, wird dieser in Tabelle 7.19 nicht separat aufgeführt.

Insgesamt konnte von den beobachteten Regeln (siehe Spalte *#Ob*) in 72,6 %, 71,43 % und 74,19 % der Fälle gezeigt werden, dass sie einen relativen Support von 100 % aufweisen. Insgesamt folgt der Programmcode somit in 72,41 %

44 Es wurde das Git-Tag *20190115* aus dem LTP-Repository verwendet.

Datentyp	#R	#No	#Ob	✓(%)	~ (%)	✗(%)
vnode	82	9	73	72,60	27,40	0,00
mount	38	7	31	74,19	25,81	0,00
buf	80	10	70	71,43	27,14	1,43

Tabelle 7.18: Zusammenfassung der Überprüfung der Sperren-Regeln in FreeBSD: Jede Zeile zeigt auf, wie viele Sperren-Regeln dokumentiert (#R), wie viele davon beobachtet (#Ob) bzw. nicht beobachtet wurden (#No). Eine Sperren-Regel bezieht sich immer auf ein Tupel aus Datentyp, Element und Zugriffstyp. Die weiteren Spalten geben jeweils den Anteil an Regeln an, die korrekt ($s_r = 1$), mehrdeutig ($0 < s_r < 1$) oder falsch ($s_r = 0$) sind.

der Fälle den dokumentierten Regeln. Bei einer ersten Kontaktaufnahme[45] mit den FreeBSD-Entwicklern signalisierten diese, dass es offenbar noch weitere Kontexte gibt, in denen die definierten Sperren-Regeln ignoriert werden dürfen. Die Sichtbarkeit der Datentypen sei nicht korrekt abgebildet[46]. Daher muss entweder die Filterliste für Funktionen erweitert werden oder aber gar der LockDoc-Ansatz selbst angepasst werden, um diese Fälle abzubilden.

Lediglich für eines der insgesamt 174 beobachteten Tupel konnte die korrekte Hypothese nicht im Datensatz gefunden werden (siehe Spalte ✗). Dieses Tupel betreffen das Lesen des Elements `b_error` der Datenstruktur `struct buf`. Aufgrund des Programmcodes ist nicht ersichtlich, ob es sich hierbei um einen Programmierfehler oder ein absichtliches Ignorieren der Sperren-Regel handelt.

Der Vergleich der obigen Analyse auf Basis der kontext-insensitiven Nachverbarbeitung mit der gleichen Analyse auf Basis der kontext-insensitiven Nachverarbeitung förderte eine interessante Beobachtung zu Tage: Das Zustandekommen von synthetischen Hypothesen durch nur einen Transaktionsstapel liefert weniger Tupel, für die die korrekte Hypothese nicht in den Daten zu finden ist (siehe Spalte ✗). Wie zuvor beschrieben, findet sich lediglich für die korrekte Hypothese für das Lesen des Elements `b_error` der Datenstruktur `struct buf` nicht in dem Datensatz. Bei der Analyse basierend auf kontext-sensitiven Nachverarbeitung sind es es jedoch insgesamt fünf Tupel. Drei Tupel betreffen das Lesen der Datenstruktur `struct buf`. Hierbei kommt auch das Element `b_error` vor. Die anderen beiden Tupel betreffen die Elemente `b_ckhashcalc` und `b_ckhash` . In beiden Fällen erfolgen die Zugriffe in anderen Kontexten als das Holen der richtigen Sperre, die hier beim Zugriff beobachtet wird. Daher kann mit der kontext-sensitiven Nachverarbeitung keine Zuordnung stattfinden. Somit kann die Regel auch nicht in dem Datensatz gefunden werden. Durch den in Abschnitt 7.3.1 beschriebenen Umweg ist dies jedoch mit der kontext-insensitiven Nachverarbeitung

45 https://lists.freebsd.org/pipermail/freebsd-fs/2019-May/027730.html
46 https://lists.freebsd.org/pipermail/freebsd-fs/2019-May/027731.html

möglich. Ein weiteres Tupel betrifft das Lesen des Elements `mnt_uppers.tqh_first`, eine verkettete Liste, der Struktur `struct mount`. Dieser Fall wurde im Detail bereits in Unterabschnitt 7.3.2 behandelt. Das letzte Tupel betrifft das Lesen des Elements `v_pollinfo`, ein Zeiger, innerhalb der Datenstruktur `struct vnode`. Eine Inspektion des Programmcodes ergab, dass die Sperre mutmaßlich nur das Beschreiben des Zeigers selbst schützt. Die Verwendung des referenzierten Speichers wird von einer anderen Sperre, die in diesem eingebettet ist, abgesichert. Auch hier ergaben Stichproben aus dem Datensatz, dass es sich bei den Zugriffen zu meist um eine Überprüfung des Zeigers auf den Wert `NULL` handelt. Hierfür scheint, wie bereits erwähnt, keine Sperre erforderlich zu sein. Daher kann die dokumentierte Sperren-Regel für das Lesen des Elements nicht den nötigen relativen Support erlangen. In allen vorgenannten Fällen sieht die korrekte Sperren-Regel nur eine Sperre vor. In vier der fünf Fälle wird zufällig die jeweils richtige Sperre auch gehalten. Da bei der kontext-insensitiven Nachverarbeitung nur ein Transaktionsstapel zum Einsatz kommt, entsteht so eine Hypothese, die u.a. die richtige Sperre enthält. Im weiteren Verlauf der Verarbeitung wird daraus die ein-Sperren-Hypothese erzeugt, die wiederum bei der oben genannten Analyse gefunden wird. Auf diese Weise kommt es bei der kontext-insensitiven Nachverbarbeitung in der letzten Spalte (✗) zu den guten Ergebnissen in Tabelle 7.18.

7.3.3.3 *Generierung von Sperren-Regeln*

Für den FreeBSD-Kern werden ebenfalls für alle Elemente der untersuchten Datentypen Sperren-Regeln abgeleitet. Eine Zusammenfassung der Ergebnisse ist in Tabelle 7.19 aufgelistet. Für das Schreiben von 64,84 % der 182 Elemente konnte jeweils eine Regel abgeleitet werden. Für das Lesen konnte in 87,36 % der Fälle eine Regel bestimmt werden. Es wurde keine Regel abgeleitet, wenn es nicht mindestens eine Beobachtung gab. Wie bereits in Unterabschnitt 7.2.4.3 ausgeführt, deutet ein geringer absoluter Support (hier: $s_a = 0$) auf eine unzureichende Arbeitslast hin. Über alle abgeleiteten Sperren-Regeln betrachtet wurde in 45,28 % der Fälle die leere Hypothese für das Lesen eines Elements ausgewählt. Der Anteil an leeren Hypothesen für den schreibenden Zugriff beträgt 15,25 %. Im Vergleich zu Linux sind dies für den lesenden Zugriff 29,01 Prozentpunkte weniger. Beim Schreiben ist der Unterschied mit 15,16 Prozentpunkten etwas geringer.

Spannend ist auch hier der Vergleich zwischen der kontext-sensitiven und kontext-insensitiven Nachverarbeitung: Bei letzterer gibt es lediglich 32,49 % leere Hypothesen. Mit der kontext-sensitiven Nachverarbeitung wird in 47,65 % der Fälle die leere Hypothese ausgewählt. Dies liegt in der Hauptsache ebenfalls an der Datenstruktur `struct buf`: Hier fällt der Anteil an leeren Hypothensen von 76,39 % mit der kontext-sensitiven Nachverarbeitung auf 22,22 %. Durch die bereits erwähnte Anpassung an die Eigenheiten von FreeBSD (vgl. Abschnitt 7.3.1) sind mit der kontext-insensitiven Nachverarbeitung deutlich mehr Speicherzugriffe mit gehaltenen Sperren zu finden. Daher steigt auch die Wahrscheinlichkeit, dass eine Hypothese einen hin-

Datentyp	#E	#Bl	#Regeln		#Nl	
			r	w	r	w
buf	45	4	37	35	10	6
bufdomain	29	18	10	6	1	0
bufqueue	6	1	2	2	0	0
fifoinfo	5	0	3	2	0	0
mount	44	3	36	24	20	9
pipepair	30	1	24	6	18	0
vnode	59	9	47	43	23	3

Tabelle 7.19: Übersicht über die abgeleiteten Sperren-Regeln für 7 Datentypen bei Verwendung der kontext-insensitiven Nachverarbeitung. Jede Zeile gibt die Anzahl der Elemente (#E), für die Regeln abgeleitet wurden, sowie die Anzahl der gefilterten Elemente (#Bl) wieder. Anschließend wird aufgeschlüsselt nach dem Zugriffstyp die Anzahl der abgeleiteten Regeln (#Regeln) aufgelistet. Die letzten Spalten geben die Summe der Elemente wieder, die keinerlei Sperren benötigen (#Nl).

reichen großen relativen Support aufweist, um als Gewinner ausgewählt zu werden.

Ein niedrigeres Grundniveau beim Anteil der leeren Hypothese an allen Hypothesen spiegelt sich auch in Abbildung 7.7 wieder. Wie schon für Linux wurde hier der Anteil der leeren Hypothese an der Gesamtheit aller Hypothesen eines Datentyps aufgetragen – unterschieden nach dem Zugriffstyp. Variiert wird hier der Schwellwert t_s im Bereich von $0,0 \leq t_s \leq 0,2$. Zunächst fällt auch bei FreeBSD ein Unterschied zwischen lesendem und schreibendem Zugriff auf. Der Anteil der leeren Hypothese für die schreibenden Zugriffe liegt erkennbar unter dem Niveau für die lesenden Zugriffe. Die Datenstrukturen `struct bufdomain`, `struct bufqueue`, `struct fifoinfo` und `struct pipepair` haben für das Schreiben von Elementen einen Anteil von 0 % leeren Hypothesen. Eine mögliche Erklärung für die Resultate ist, dass beim Schreiben konsequent auf die Einhaltung der Sperren-Regeln geachtet wird. Beim Lesen von Elementen wird nur dann eine Sperre eingesetzt, wenn die Konsistenz von Bedeutung ist. Spielt die Konsistenz der Daten eine untergeordnete Rolle, würde der Einsatz von Sperren den möglichen Grad der Parallelität einschränken. Dies ist für Mehrkernbetriebssysteme nicht sinnvoll. Diese Fragestellung kann jedoch nur durch weitere Kommunikation mit den FreeBSD-Entwicklern als Domänenexpertern aufgelöst werden.

Auch in FreeBSD gibt es Muster, die sich nicht ohne Weiteres in LockDoc abbilden lassen und zu verfälschten Ergebnissen führen: Die Datenstruktur `struct vnode` verfügt ebenfalls über einen atomar-zugreifbaren Referenzzähler. Analog zum Linux-Kern werden Transitionen $1 \rightarrow 0$ bzw. $0 \rightarrow 1$ mit

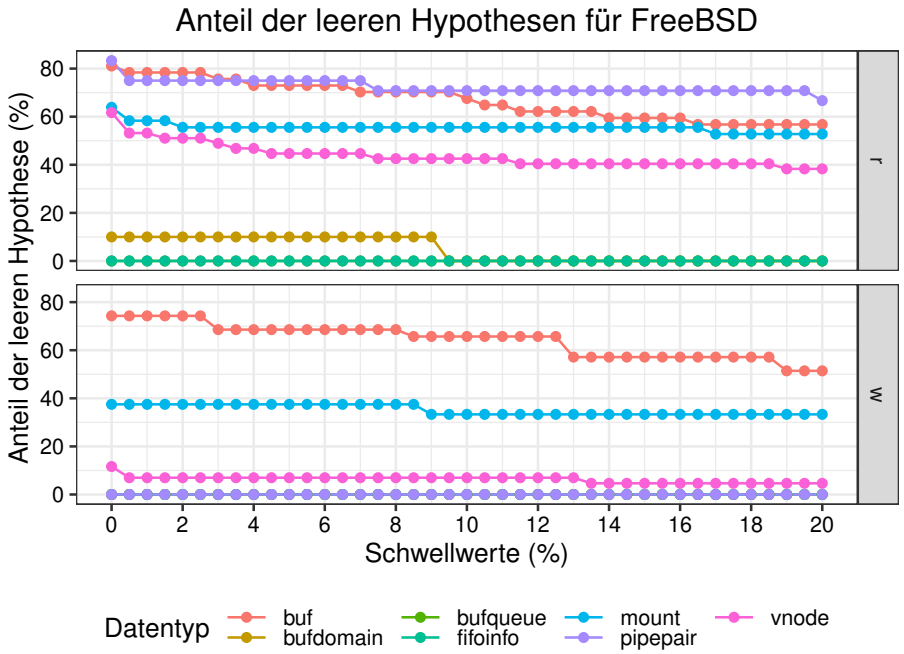


Abbildung 7.7: Aufgetragen wird der Anteil der leeren Hypothese („keine Sperre nötig“) an allen Hypothesen für jeden Datentyp in FreeBSD für verschiedene Werte von t_s – unterschieden nach dem Zugriffstyp. Es wurde die Strategie *Sharpen* sowie die kontext-insensitive Nachverarbeitung verwendet.

einer Sperre abgesichert. Anders als in Linux ist dieses Vorgehen in FreeBSD explizit dokumentiert[47].

7.3.3.4 *Verstöße gegen Sperren-Regeln*

Auch in FreeBSD werden für jede abgeleitete Sperren-Regel mit $s_r < 1$ Zugriffe identifiziert, die nicht mit dieser übereinstimmen. Tabelle 7.20 gibt eine Übersicht dieser Fälle für den FreeBSD-Kern. Mit insgesamt 32.585 Gegenbeispielen ist die absolute Anzahl geringer als die von Linux. Sie ist jedoch höher als die Anzahl Gegenbeispiele für die kontext-sensitive Nachverarbeitung. Hier finden sich 1.562 Gegenbeispiele. Da, wie im vorherigen Unterabschnitt 7.3.3.3 beschrieben, mit der kontext-insensitiven Nachverarbeitung für mehr Tupel eine Sperren-Regel bestimmt wird, die nicht der leeren Regel entspricht, kann es auch mehr Gegenbeispiele geben. Denn anders als die leere Hypothese kann die Gewinnerhypothese einen relativen Support kleiner eins aufweisen. Insgesamt sind beide Werte zu hoch, als dass es hierbei bei jedem einzelnen um einen echten Synchronisationsfehler handeln kann. Ein Grund kann die bereits in Unterabschnitt 7.3.3.2 angesprochene Schwäche seitens LockDoc sein, die Sichtbarkeit von Objekten in FreeBSD nicht korrekt abzubilden. Dies würde zu falsch-positiven Gegenbeispielen führen, obwohl die Zugriffe in den betreffenden Kontexten keinerlei Nebenläufigkeit befürchten müssen.

47 Siehe `freebsd/sys/sys/vnode.h`, Zeile 82.

Datatyp	Häufigkeit	Elemente	Kontexte
buf	4457	4	38
bufdomain	19416	3	39
mount	2502	3	17
pipepair	6	2	3
vnode	6204	8	112

Tabelle 7.20: Übersicht über die Gegenbeispiele zu den abgeleiteten Sperren-Regeln unter FreeBSD für die kontext-insensitive Nachverarbeitung. Pro Zeile wird die Anzahl der falsch-abgesicherten Zugriffe, die zugehörigen Elemente sowie die Kontexte aufgelistet. Insgesamt gab es 32.585 Zugriffe 209 aus Kontexten. Datentypen ohne Gegenbeispiele wurden ausgelassen.

Eine manuelle Inspektion der Gegenbeispiele zeigte, dass die Resultate für die Datenstruktur `struct buf` falsch-positiv sind. Hier wurde aufgrund eines zu großen Schwellenwertes im Fall des Elements `b_cluster` eine andere Hypothese als Gewinner bestimmt. Die richtige Hypothese hat einen relativen Support von 100 %. Für das Element `b_dep` ist eine implementierungsspezifische Sperre erforderlich. Hier ist das zugehörige Subsystem bisher mit LockDoc noch nicht abgedeckt. Im Fall des Elements `b_bufobj` suggeriert der Programmcode, dass es legitim sei, keinerlei Sperren einzusetzen. Der letzte Fall betrifft das Element `b_subqueue`, was bereits in Unterabschnitt 7.3.2 diskutiert wurde. Die Gegenbeispiele der Datenstruktur `struct bufdomain` sind ebenfalls falsch-positv, da hier jeweils eine andere Hypothese mit 100 % relativem Support aber mit mehr Sperren ausgewählt wurde. Bei der Struktur `struct mount` handelt es sich in einem Fall um einen absichtlich zu Testzwecken von dem Autor dieser Arbeit eingefügten Fehler. Die Gegenbeispiele für das Element `mnt_iosize_max` können nicht abschließend bewertet werden, da laut Dokumentation[48] lediglich eine Referenz auf das zugehörige Objekt nötig ist. Referenzzähler sind jedoch noch nicht in LockDoc abgebildet. Die Gegenbeispiele zu dem Element `mnt_secondary_writes` sind unklar. Laut Aussage der Entwickler handelt es sich bei diesem Zugriff um einen unproblematischen Zugriff: „[...] and the possible race is innocent.“[49]. Gemäß der Idee von LockDoc ist dies ein valides Gegenbeispiel, das aber als ungefährlich einzustufen ist. Ähnlich stellt sich die Situation für den Datentyp `struct pipepair` dar: Bei dem einen Element handelt es sich ebenfalls um einen absichtlichen Fehler. Die Gegenbeispiele zu dem anderen Element scheinen ein absichtliches Ignorieren der Sperren-Regeln zu sein. Die betreffende Funktion ist mit folgendem Kommentar versehen: „We shouldn’t need locks here as we’re doing a read and this should be a natural race.“[50]. Die Gegenbeispiele zu der Da-

48 Siehe `sys/sys/mount.h`, Zeilen 191 und 216.
49 http://docs.FreeBSD.org/cgi/mid.cgi?YG99ecPrPeom/sGw
50 Siehe `sys/kern/sys_pipe.c`, Zeile 1496 ff.

tenstruktur `struct vnode` enthalten ebenfalls einen absichtlichen Fehler. Darüber hinaus wird für das Lesen der Elemente `v_bufobj.bo_clean.bv_root` und `v_bufobj.bo_dirty.bv_root` mal die Leser- und mal die Schreiber-Sperre der betreffenden Leser-Schreiber-Sperre gehalten. Daher kann sich keine der beiden Hypothesen durchsetzen. Somit wird eine andere, falsche Hypothese als Gewinner ausgewählt.

Für das Element `v_bufobj.bo_ops` sind hingegen eigentlich keine Sperren erforderlich, dennoch wird eine nicht-leere Hypothese ausgewählt. Es ist zu überlegen, ob dieses Element im Rahmen der Nachverarbeitung gefiltert werden sollte. Die Gegenbeispiele für das Lesen und Schreiben des Elements `v_hashlist.le_next` sind nicht ganz eindeutig. Es handelt sich hierbei um ein ähnliches Szenario wie für das Element `i_hash` aus Unterabschnitt 7.2.4.3: Das Element wird benötigt, um eine Instanz in eine verkettete Liste einzufügen. Dementsprechend gibt es bei Operationen immer zwei beteiligte Instanzen. Aus der Sicht der zugegriffenen Instanz wird aber die eigene und eine in anderen Instanz eingebettete Sperre gehalten. Die Regel schreibt in diesem Fall zwei Sperren, `EMBSAME(vnode.v_lock[w])-> vfs_hash_lock(rw[w])`, vor. Über alle Zugriffe hinweg wird aber nur die globale Sperre `vfs_hash_lock(rw[w])` gehalten. Der betreffende Programmcode[51] lässt den Schluss zu, dass mutmaßlich nur die globale Sperre nötig ist. Mit letzter Gewissheit kann dies jedoch nur mit Hilfe von Experten geklärt werden. Das Gegenbeispiel zu dem Element `v_irflag` ist auch falsch-positiv, da die richtige Hypothese ebenfalls 100 % relativen Support hat. Der Schwellenwert ist auch hier zu groß gewählt. Die Gegenbeispiele zu dem Element `v_type` sind mutmaßlich falsch-positiv, da laut Dokumentation[52] nur der Referenzzähler erhöht werden muss. Die verdächtigen Zugriffe auf das Element `v_vflag` müssen eventuell gefiltert werden, da die zugreifende Funktion die betreffende Instanz gerade erst alloziert hat.

7.3.4 *Zusammenfassung*

In diesem Abschnitt wurde die Anpassung des LockDoc-Ansatzes auf den FreeBSD-Kern sowie deren Evaluation vorgestellt. Auch hier sehen die Ergebnisse in Summe vielversprechend aus: Die in Unterabschnitt 7.3.2 gewählte Herangehensweise, Gegenbeispiele für die dokumentierte Sperren-Regel anstatt der abgeleiteten Sperren-Regel zu bestimmen, scheint aussichtsreich zu sein. Dies sollte in zukünftigen Arbeiten vertieft werden. Das Überprüfen der Sperren-Dokumentation zeigte in 72,41 % der Fälle eine Übereinstimmung zwischen Dokumentation und dem beobachtetem Verhalten und fällt somit besser aus als die gleiche Untersuchung für Linux. Zukünftig sind in diese Untersuchung noch weitere Datentypen einzubeziehen, um die Quantität zu verbessern.

Im Gegensatz zu Linux konnte für FreeBSD detailliert aufgeschlüsselt werden, warum die Sperren-Regeln in einem bzw. fünf Fällen nicht im Datensatz

51 Siehe `sys/kern/vfs_hash.c`.
52 Siehe `sys/sys/vnode.h`, Zeile 107.

aufzufinden waren. Dies führte außerdem zu einer Änderung an der existierenden Sperren-Dokumentation im FreeBSD-Code. Im Vergleich zu Linux ist die Anzahl der Gegenbeispiele mit 32.585 Fällen geringer, wenngleich für FreeBSD weniger Datenstrukturen untersucht wurden. Festzuhalten ist dennoch, dass der Umfang insgesamt gering genug ist, um eine manuelle Inspektion durchzuführen und alle Fälle einzeln zu bewerten. Für (fast) alle Fälle gelang es, eine plausible Erklärung zu finden. Dennoch wurde auch für FreeBSD deutlich, dass weitere Kommunikation mit den Entwicklern erforderlich ist, um die Abläufe im Kern besser in LockDoc abzubilden. Außerdem muss das Ziel zukünftiger Arbeiten sein, einen vergleichbaren Umweg für die Datenstruktur `struct buf` zu finden, so dass auch die kontext-sensitive Nachverarbeitung funktioniert. Denn die hier vorgestellten Stichproben lassen vermuten, dass sie genauere Ergebnisse liefert. Abschließend ist noch auf die beiden Änderungen am FreeBSD-Programmcode zu verweisen, die durch die Untersuchungen im Rahmen dieser Arbeit angestoßen wurden: In einem Fall wurde die Sperren-Dokumentation aktualisiert. In einem anderen Fall wurde ein unsicherer Zugriff mit den entsprechenden Sperren versehen.

8

ZUSAMMENFASSUNG UND AUSBLICK

Inhalt

Die über vergangenen Jahrzehnte gestiegene Anzahl der Prozessorkerne pro Chip stellen wachsende parallele Rechenleistung bereit – vgl. Kapitel 1. Um diese Leistung auch in Betriebssystemen nutzbar zu machen, verschob sich das Verhalten von einem grobgranularen Sperren hinzu einem feingranularen auf der Ebene von Datenstrukturen oder teilen davon – vgl. Kapitel 3. Da mehr als eine einzige Sperre beteiligt ist, muss z. B. die korrekte Reihenfolge beim Belegen der Sperren eingehalten werden, um Verklemmungen zu vermeiden. Dieses Problem sowie weitere können im schlimmsten Fall zum Absturz des Betriebssystem oder zu einer Datenkorruption führen – vgl. Kapitel 2. Somit kommt dem korrekten synchronisierten Zugriff auf Daten eine entsprechende Bedeutung zu. Um dies sicherstellen zu können, muss es jedoch eine Sperren-Dokumentation zur Überprüfung geben. Wie die Analyse in Kapitel 3 jedoch zeigte, ist dies nur bedingt der Fall. Mit der Analyse der existierenden Forschungsarbeiten in Kapitel 4 ergeben sich daraus zwei zentrale Probleme:

PROBLEM 1: Die existierende Sperren-Dokumentation in Betriebssystemkernen folgt keiner präzisen Syntax. Sie ist außerdem über den Quellcode verstreut und teilweise unvollständig oder veraltet.

PROBLEM 2: Existierende Forschungsarbeiten befassen sich zwar mit dem Auffinden von verschiedensten Synchronisationsfehlern. Sie lassen aber Ansätze zum Rückführen der Erkenntnisse in Form von neuer Dokumentation vermissen.

Daraus folgend kann man die Forschungsfragen dieser Arbeit nochmal in einer Frage bündeln: „Wie kann man mit der aufzeichnungsbasierten Analyse Erkenntnisse über das Synchronisationsverhalten in Mehrkernbetriebssystemen den Entwicklern nutzbar machen?“ Die dazu gehörenden Forschungsbeiträge werden in dem folgenden Abschnitt 8.1 nochmal zusammengefasst. Der abschließende Abschnitt 8.2 gibt einen Ausblick über die weiterführenden Arbeiten.

8.1 FORSCHUNGSBEITRÄGE

Die Ergebnisse dieser Arbeit bringen den aktuellen Stand der Kunst durch die folgenden drei Beiträge voran:

FORSCHUNGSBEITRAG 1: Der erste Forschungsbeitrag dieser Arbeit ist der LockDoc-Ansatz zur Sperrenanalyse von Softwaresystemen [LSBS19]. Der Fokus des Ansatzes liegt auf der Analyse von Betriebssystemkernen. Während der Ausführung einer Arbeitslast werden Speicherzugriffe sowie Sperren-Operationen in dem zu untersuchenden Betriebssystem aufgezeichnet. Die Nachverarbeitung der Aufzeichnung überführt die Daten angereichert mit Informationen aus dem Quelltext in eine relationale Darstellung. Auf diese Weise werden die Zusammenhänge zwischen Datenstrukturen, Instanzen ebendieser, Zugriffe darauf und den Sperren festgehalten. Dies und der zeitliche Verlauf der Ereignisse sind so für eine weitere Verarbeitung zugreifbar.

FORSCHUNGSBEITRAG 2: Der zweite Forschungsbeitrag fokussiert sich auf die Verwendung der erhobenen Daten: Zunächst lässt sich anhand des aufgezeichneten Verhaltens überprüfen, ob der Programmcode noch der existierenden Sperren-Dokumentation folgt. Ist dies nicht der Fall oder die existierende Dokumentation ist unvollständig, so können aus den Daten neue Sperren-Regeln für Datenstrukturen abgeleitet und daraus eine systematische Sperren-Dokumentation für alle beobachteten Datentypen generiert werden. Als dritter und letzter Bestandteil der Auswertung dienen die abgeleiteten Regeln als Ausgangspunkt zum Auffinden von Gegenbeispielen innerhalb der Aufzeichnung, die nicht den Sperren-Regeln folgen. Dies hilft den Entwicklern unmittelbar die Softwarequalität zu verbessern, da sie direkt Fehler beheben können. Die zwei zuvor genannten Punkte sorgen mittel- bis langfristig für eine bessere Qualität. Die Entwickler werden durch eine bessere Dokumentation in die Lage versetzt, von vornherein Synchronisationsfehler bei der Entwicklung zu vermeiden [LSBS19].

FORSCHUNGSBEITRAG 3: Beim dritten Forschungsbeitrag wurden die Betriebssystemkerne von Linux [LSBS19] und FreeBSD unter den drei zuvor genannten Gesichtspunkten untersucht. In beiden Fällen zeigte sich, dass der Programmcode nur in 58,28 % bzw. 72,41 % der Fälle die Sperren-Dokumentation zu 100 % befolgt. Für beide Betriebssystemkerne konnten insgesamt für 40 Datentypen Sperren-Regeln abgeleitet werden. Wobei in 74,29 % für Linux und 45,28 % für FreeBSD der Fälle die leere Hypothese gewann. Die Auswertung der Gegenbeispiele führte in einem Fall zu einer akzeptierten Änderung am Linux-Kern. Außerdem wurden mehre Änderung angenommen, die die Sperren-Dokumentation aufgrund der Ergebnisse von LockDoc korrigierten. In Summe wurden fünf Änderungen an dem Linux-Kern eingereicht und von den Entwicklern akzeptiert. Für FreeBSD führten die Ergebnisse zu zwei Änderungen, die durch die FreeBSD-Entwickler vorgenommen wurden. Diese betrafen die Sperren-Dokumentation sowie einen echten Programmierfehler.

8.2 AUSBLICK

Während der Arbeit an den vorgenannten Forschungsbeiträgen ergaben sich weitere Ideen für anschließende Forschungsarbeiten, um an verschiedenen Stellen des LockDoc-Ansatzes anzuknüpfen. Im Folgenden werden die Ideen unterteilt nach drei Gruppen dargestellt.

ARBEITSLAST Bei der Verbesserung der Arbeitslast ist das oberste Ziel, die Basisblockabdeckung zu erhöhen, um möglichst viel Programmcode auszuführen. Hierzu kann untersucht werden, wie die mit Hilfe des Werkzeugs *syzkaller* generierten Programme in die existierende Arbeitslast von LockDoc integriert werden können. Außerdem kann es lohnenswert sein zu untersuchen, inwieweit sich der genetische Algorithmus von *syzkaller* modifizieren lässt, um noch mehr Abdeckung in dem jeweiligen Subsystem zu erzielen. Abschließend bleibt noch die Evaluation des Ansatzes aus Unterabschnitt 5.1.2.2 unter FreeBSD. Für FreeBSD lohnt sich ebenfalls die Evaluation der im FreeBSD-Repository bereits enthaltenen Regressions-[1] und Stress-Tests[2].

ANALYSE In der Analyse-Phase von LockDoc werden die Zugriffe innerhalb von Transaktionen ausgewertet, um daraus Sperren-Regeln abzuleiten. Hier wird bisher nur zwischen drei Arten von Sperren unterschieden: globale Sperren, in die zugegriffene Datenstruktur oder in irgendeine andere Datenstruktur eingebettete Sperren. Dies lässt sich jedoch verfeinern: Zwischen Instanzen von Datenstrukturen gibt es nicht selten Beziehungen, wie z. B. der Verweis einer Inode auf den zugehörigen Superblock. Dies kann ausgenutzt werden, um beispielsweise die unspezifische Regel `EMBOTHER(super_block.sb_lock)` für den Zugriff auf eine Inode zu `EMSAME(inode.i_sb.sb_lock)` zu verfeinern.

Für die Bewertung einer Hypothese wird bisher der relative Support s_r herangezogen. Die Aufrufhierarchie, die zu einem Zugriff führte, wird bisher nicht betrachtet. Diese könnte jedoch eingesetzt werden, um jede Hypothese mit einer Konfidenz zu versehen. Gibt es eine große Diversität in der Aufrufhierarchie, resultiert dies in einer hohen Konfidenz. Andernfalls wirkt eine Hypothese weniger vertrauenswürdig. Eine andere Idee zur Bewertung von Hypothesen berücksichtigt andere innerhalb einer Transaktion zugegriffene Variablen: Die daraus abgeleiteten Zugriffsmuster, wie z. B. „immer/meistens wenn X gelesen wurde, wird auch Y gelesen“, können auf die Hypothesen übertragen wurden. Widersprechen sich die Gewinnerhypothesen für zwei Elemente, so ist das Vertrauen in diese Hypothesen ebenfalls gering. Andernfalls kann das Ergebnis als plausibel betrachtet werden. Hier kann ggf. die Arbeit von Lu et al. zu Rate gezogen werden [LPH+07].

1 https://github.com/freebsd/freebsd-src/tree/main/tests/sys
2 https://github.com/freebsd/freebsd-src/tree/main/tools/test/stress2

FALLSTUDIEN Die durchgeführten Untersuchungen an dem Betriebssystemkern von Linux und FreeBSD zeigten auf, an welchen Stellen LockDoc die Realität in echten Betriebssystemen noch nicht korrekt abbildet. Dies kann hier in der Zukunft durch intensivere Kommunikation mit den Kernentwicklern verbessert werden. Dazu zählt insbesondere ein besseres Verständnis, warum die Ergebnisse von LockDoc noch nicht zu der Realität passen. Ein konkreter Punkt ist u.a. das korrekte Abbilden der Sichtbarkeit von Objekten in FreeBSD. In Linux hingegen müssen Ausführungskontexte erkannt werden, die aufgrund von nicht-vorhandener Parallelität keinerlei Synchronisationsmechanismen bedürfen. Darüber hinaus können die bereits existierenden Datensätze weiter genutzt werden, um nach Gegenbeispielen für die bereits dokumentierten Sperren-Regeln zu suchen – sofern sich die Dokumentation als aktuell und glaubwürdig erweist. Im Fall von FreeBSD führte dieses Vorgehen bereits zu einem entdeckten Programmierfehler (vgl. Unterabschnitt 7.3.2). In den Unterabschnitt 7.2.4.4 sowie Unterabschnitt 7.3.3.4 wurden bisher die Gegenbeispiele auf Basis der abgeleiteten Sperren-Regeln bestimmt.

ANHANG

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

ALGORITHMENVERZEICHNIS

[AASEH17] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, 8(1):4, 2017. doi: 10.1186/s13174-017-0055-2. (Auf den Seiten 1 und 23 zitiert.)

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018. (Auf den Seiten 23 und 26 zitiert.)

[AHMN91] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 234–243, New York, NY, USA, 1991. ACM Press. doi: 10.1145/115952.115976. (Auf Seite 14 zitiert.)

[And90] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan 1990. doi: 10.1109/71.80120. (Auf Seite 26 zitiert.)

[ÅNK11] Mikael Åsberg, Thomas Nolte, and Shinpei Kato. A loadable task execution recorder for hierarchical scheduling in Linux. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 380–387. IEEE, aug 2011. doi: 10.1109/rtcsa.2011.28. (Auf den Seiten 4 und 50 zitiert.)

[ASo6] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging - PADTAD '06*. ACM Press, 2006. doi: 10.1145/1147403.1147413. (Auf den Seiten 4, 48 und 51 zitiert.)

[Balo2] John H. Baldwin. Locking in the multithreaded freebsd kernel. In *Proceedings of the BSDCon '02 Conference on File and Storage Technologies*, pages 27–36, Cathedral Hill Hotel, San Francisco, California, USA, February 2002. (Auf Seite 49 zitiert.)

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 29–44, New York, NY, USA, 2009. ACM. doi: 10.1145/1629575.1629579. (Auf Seite 11 zitiert.)

[BC05] Daniel Pierre Bovet and Marco Cesati. *Understanding The Linux Kernel*. O'Reilly Media Inc., 3rd edition, November 2005. (Auf den Seiten 32, 54 und 92 zitiert.)

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC '05)*, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association. (Auf Seite 50 zitiert.)

[Bev89] W. R. Bevier. Kit: a study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989. (Auf Seite 42 zitiert.)

[BH00] Ray Bryant and John Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the Linux® kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference – Volume 4*, ALS '00, pages 17–17, Atlanta, Georgia, October 2000. USENIX Association. (Auf den Seiten 4, 48 und 51 zitiert.)

[BJV03] Fred Barnes, Christian Jacobsen, and Brian Vinter. RMoX: A raw-metal occamplain experiment. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 269–288, Amsterdam, September 2003. IOS Press. (Auf Seite 42 zitiert.)

[BLCH19] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 255–268, Renton, WA, July 2019. USENIX Association. (Auf den Seiten 4, 45 und 51 zitiert.)

[BP84] Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation. *Commun. ACM*, 27(1):42–52, January 1984. doi: 10.1145/69605.2085. (Auf Seite 3 zitiert.)

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, January 2002. doi: 10.1145/565816.503274. (Auf den Seiten 4, 46 und 51 zitiert.)

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM Press. doi: 10.1145/224056.224077. (Auf Seite 42 zitiert.)

[Bus90] David Bustard. Concepts of concurrent programming. Technical report, Carneige Mellon University, Software Engineering Institute, 1990. (Auf Seite 23 zitiert.)

[BV04] Peter T. Breuer and Marisol García Valls. Static deadlock detection in the Linux kernel. In *Lecture Notes in Computer Science*, pages 52–64. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-24841-5_4. (Auf den Seiten 4, 46, 51 und 78 zitiert.)

[CB08] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Queue*, 6(5):16–25, September 2008. doi: 10.1145/1454456.1454462. (Auf den Seiten 1, 27 und 28 zitiert.)

[CBJ+19] Q. Chen, J. Bai, Z. Jiang, J. Lawall, and S. Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, Feb 2019. doi: 10.1109/SANER.2019.8668017. (Auf den Seiten 4, 48 und 51 zitiert.)

[CD11] Andrea Claudi and Aldo Franco Dragoni. Testing Linux-based real-time systems: Lachesis. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, 2011. (Auf Seite 57 zitiert.)

[CH93] M. D. Campbell and R. L. Holt. Lock-granularity analysis tools in svr4/mp. *IEEE Software*, 10(2):66–70, March 1993. doi: 10.1109/52.199738. (Auf den Seiten 27 und 28 zitiert.)

[CMS+17] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM Press, 2017. (Auf Seite 58 zitiert.)

[Cor06] Jonathan Corbet. The kernel lock validator, March 2006. (Auf den Seiten 4, 49, 51 und 62 zitiert.)

[CS12] Guancheng Chen and Per Stenstrom. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and*

Analysis, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press. (Auf den Seiten 4, 48 und 51 zitiert.)

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM Press. doi: 10.1145/502034.502042. (Auf den Seiten 1 und 3 zitiert.)

[CZC+15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM Press. doi: 10.1145/2815400.2815402. (Auf den Seiten 4 und 42 zitiert.)

[Dav16] David Drysdale. Coverage-guided kernel fuzzing with syzkaller. https://lwn.net/Articles/677764/, March 2016. Zugegriffen: 13.11.2020. (Auf Seite 57 zitiert.)

[DGR+74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974. doi: 10.1109/JSSC.1974.1050511. (Auf Seite 1 zitiert.)

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965. doi: 10.1145/365559.365617. (Auf Seite 14 zitiert.)

[Dij68a] Edsger W. Dijkstra. The structure of the "the"-multiprogramming system. *Commun. ACM*, 11(5):341–346, May 1968. doi: 10.1145/363095.363143. (Auf Seite 17 zitiert.)

[Dij68b] E.W. Dijkstra. Cocoperating sequential processes. In *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys*, pages 43–112, United States, 1968. Academic Press Inc. (Auf den Seiten 15, 16 und 18 zitiert.)

[dOCdO19] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Efficient formal verification for the Linux kernel. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 315–332, Cham, 2019. Springer International Publishing. (Auf den Seiten 4 und 42 zitiert.)

[Dow16] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2.2.1 edition, 2016. (Auf Seite 15 zitiert.)

[EA03] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM. doi: 10.1145/945445.945468. (Auf den Seiten 4, 43 und 51 zitiert.)

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011. (Auf Seite 1 zitiert.)

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM. doi: 10.1145/502034.502041. (Auf den Seiten 4, 43, 51 und 62 zitiert.)

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the*

Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM Press. doi: 10.1145/224056.224076. (Auf Seite 11 zitiert.)

[EMBO10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 151–162, USA, 2010. USENIX Association. (Auf den Seiten 4, 48 und 51 zitiert.)

[FLH+15] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311, 09 2015. doi: 10.1093/bioinformatics/btv557. (Auf Seite 81 zitiert.)

[FRB14] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, Broomfield, CO, October 2014. USENIX Association. (Auf Seite 50 zitiert.)

[Gee05] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005. doi: 10.1109/MC.2005.160. (Auf Seite 1 zitiert.)

[H+] Cyril Hrubis and others. Linux test project. https://github.com/linux-test-project/ltp. Accessed: 2020-08-20. (Auf den Seiten 55, 101 und 121 zitiert.)

[HCC+12] Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, and Mingyu Chen. HaLock. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*. ACM Press, 2012. doi: 10.1145/2370816.2370854. (Auf den Seiten 4, 48 und 51 zitiert.)

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM Press. doi: 10.1145/268998.266660. (Auf Seite 13 zitiert.)

[HI13] Wim H. Hesselink and Mark IJbema. Starvation-free mutual exclusion with semaphores. *Formal Aspects of Computing*, 25(6):947–969, 2013. doi: 10.1007/s00165-011-0219-y. (Auf Seite 25 zitiert.)

[Hin01] Michael Hind. Pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '01*. ACM Press, 2001. doi: 10.1145/379605.379665. (Auf den Seiten 42 und 43 zitiert.)

[HK13] Shin Hong and Moonzoo Kim. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software*, 86(2):377 – 388, 2013. doi: https://doi.org/10.1016/j.jss.2012.08.063. (Auf Seite 43 zitiert.)

[HL07] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 265–280, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. (Auf Seite 43 zitiert.)

[HLX+18] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 200–210, New York, NY, USA, 2018. ACM Press. doi: 10.1145/3196321.3196334. (Auf Seite 41 zitiert.)

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974. doi: 10.1145/355620.361161. (Auf den Seiten 18 und 19 zitiert.)

[IF10] Ayelet Israeli and Dror G. Feitelson. The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485 – 501, 2010. doi: 10.1016/j.jss.2009.09.042. (Auf Seite 2 zitiert.)

[Iye02] Manoj Iyer. Analysis of Linux Test Project's kernel code coverage. http://kernel.poly.ro/2.4/kernel%20docs/kernel_coverage.pdf, July 2002. (Auf Seite 57 zitiert.)

[J+06] David Jones and others. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, March 2006. Zugegriffen am 13.11.2020. (Auf Seite 57 zitiert.)

[Jak16] Jake Edge. Fuzzing filesystems with afl. https://lwn.net/Articles/685182/, April 2016. Zugegriffen: 13.11.2020. (Auf Seite 57 zitiert.)

[JKS+19] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, may 2019. doi: 10.1109/sp.2019.00017. (Auf den Seiten 4, 49, 50, 54 und 82 zitiert.)

[Jon04] Jonathan Corbet. Finding kernel problems automatically. https://lwn.net/Articles/87538/, June 2004. Zugegriffen: 19.05.2020. (Auf den Seiten 4 und 46 zitiert.)

[Jon14a] Jonathan Corbet. Mcs locks and qspinlocks. https://lwn.net/Articles/590243/, March 2014. Zugegriffen: 27.02.2020. (Auf den Seiten 26, 27 und 28 zitiert.)

[Jon14b] Jonathan Corbett. The big kernel lock lives on. https://lwn.net/Articles/86859/, March 2014. Zugegriffen: 04.03.2020. (Auf den Seiten 1 und 31 zitiert.)

[JPH+09] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 24–35, New York, NY, USA, 2009. ACM Press. doi: 10.1145/1516360.1516365. (Auf Seite 26 zitiert.)

[JTZC08] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 295–308, USA, 2008. USENIX Association. (Auf den Seiten 4, 48 und 51 zitiert.)

[JYX+16] Yunyun Jiang, Yi Yang, Tian Xiao, Tianwei Sheng, and Wenguang Chen. DRD-DR: a lightweight method to detect data races in Linux kernel. *The Journal of Supercomputing*, 72(4):1645–1659, mar 2016. doi: 10.1007/s11227-016-1691-1. (Auf den Seiten 4, 48 und 51 zitiert.)

[KEH+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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. doi: 10.1145/1629575.1629596. (Auf den Seiten 4 und 42 zitiert.)

[Kle09] Gerwin Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, feb 2009. doi: 10.1007/s12046-009-0002-4. (Auf Seite 42 zitiert.)

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, San Francisco, CA, USA, 09 2019. (Auf den Seiten 4 und 42 zitiert.)

[LAC+15] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on*

Programming Languages and Operating Systems, PLOS '15, pages 21–26, New York, NY, USA, 2015. ACM Press. doi: 10.1145/2818302.2818306. (Auf den Seiten 4 und 42 zitiert.)

[Lam86a] Leslie Lamport. The mutual exclusion problem: Part i - a theory of interprocess communication. *J. ACM*, 33(2):313–326, April 1986. doi: 10.1145/5383.5384. (Auf Seite 14 zitiert.)

[Lam86b] Leslie Lamport. The mutual exclusion problem: Partii - statement and solutions. *J. ACM*, 33(2):327–348, April 1986. doi: 10.1145/5383.5385. (Auf Seite 14 zitiert.)

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, January 1989. doi: 10.1145/63238.63240. (Auf Seite 23 zitiert.)

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, dec 1992. doi: 10.1145/161494.161501. (Auf Seite 43 zitiert.)

[Lar02] Paul Larson. Testing Linux with the Linux Test Project. In *Proceedings of the Ottawa Linux Symposium*, pages 265–273, 2002. (Auf den Seiten 55 und 57 zitiert.)

[Law96] Kevin P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux Journal*, 1996(29):7, September 1996. (Auf Seite 72 zitiert.)

[LBP19] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS'19, pages 8–15, New York, NY, USA, 2019. ACM Press. doi: 10.1145/3365137.3365395. (Auf den Seiten 4 und 42 zitiert.)

[LCH+12] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution mining. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments - VEE '12*. ACM Press, 2012. doi: 10.1145/2151024.2151044. (Auf den Seiten 4, 50, 52 und 54 zitiert.)

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM Press. doi: 10.1145/1065010.1065034. (Auf den Seiten 4 und 50 zitiert.)

[Le91] Tin Le. tsys. https://groups.google.com/g/alt.sources/c/V_B37EtnWKQ/m/NztsljVYV84J, September 1991. Zugegriffen am 13.11.2020. (Auf Seite 57 zitiert.)

[Leh01] Greg Lehey. Improving the freebsd smp implementation. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 155–164, 2001. (Auf den Seiten 1 und 35 zitiert.)

[LHRF03] Paul Larson, Nigel Hinds, Rajan Ravindran, and Hubertus Franke. Improving the Linux Test Project with kernel code coverage analysis. In *Proceedings of the Ottawa Linux Symposium*, pages 275–289, 2003. (Auf Seite 57 zitiert.)

[Lie93] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM Press. doi: 10.1145/168619.168633. (Auf Seite 13 zitiert.)

[Lie95] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM Press. doi: 10.1145/224056.224075. (Auf den Seiten 11 und 13 zitiert.)

[Lie96] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, September 1996. doi: 10.1145/234215.234473. (Auf den Seiten 11 und 13 zitiert.)

[Lin04] Linus Torvalds. Sparse "context" checking. https://lwn.net/Articles/109066/, October 2004. Zugegriffen: 19.05.2020. (Auf den Seiten 4, 46, 47 und 51 zitiert.)

[Lin18] Linux Community. Sparse wiki. https://sparse.wiki.kernel.org/index.php/Main_Page, December 2018. Zugegriffen: 19.05.2020. (Auf Seite 46 zitiert.)

[Lin19] Linux Kernel. Sparse. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/dev-tools/sparse.rst?h=v5.6, July 2019. Zugegriffen: 19.05.2020. (Auf Seite 47 zitiert.)

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *SIGPLAN Not.*, 42(6):278–289, June 2007. doi: 10.1145/1273442.1250766. (Auf Seite 43 zitiert.)

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Wesley, Boston, MA, USA, 3rd edition, June 2010. (Auf den Seiten 1, 12, 13, 15, 17, 20, 22, 23, 27, 28, 31, 32, 54, 60, 76, 92, 93 und 105 zitiert.)

[LPH+07] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP '07*. ACM Press, 2007. doi: 10.1145/1294261.1294272. (Auf den Seiten 4, 44, 51, 64 und 131 zitiert.)

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM Press. doi: 10.1145/1346281.1346323. (Auf den Seiten 1, 23, 24 und 25 zitiert.)

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, February 1980. doi: 10.1145/358818.358824. (Auf Seite 18 zitiert.)

[LSBS19] **Alexander Lochmann**, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-based analysis of locking in the Linux kernel. In *Proceedings of the 14th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '19)*, New York, NY, USA, March 2019. ACM Press. doi: 10.1145/3302424.3303948. (Auf den Seiten v, 3, 8, 9, 29, 32, 33, 34, 35, 38, 39, 41, 53, 61, 62, 63, 64, 65, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 92, 93, 94, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113 und 130 zitiert.)

[LSC97] Wenke Lee, Salvatore Stolfo, and Philip K. Chan. Learning patterns from Unix process execution traces for intrusion detection. In *Proceedings of the Workshop on AI Approaches to Fraud Detection and Risk Management*, AAAI, Providence, RI, USA, July 1997. AAAI Press. doi: 10.7916/D8B56RF2. (Auf den Seiten 4 und 50 zitiert.)

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993. doi: 10.1109/MC.1993.274940. (Auf Seite 2 zitiert.)

[LTS20] **Alexander Lochmann**, Robin Thunig, and Horst Schirmeier. Improving Linux-kernel tests for lockdoc with feedback-driven fuzzing. In *Tagungsband des FG-BS Herbsttreffens 2020*, Bonn, 2020. Gesellschaft für Informatik e.V.z. doi: 10.18420/fgbs2020h-01. (Auf den Seiten v, 8, 9, 55, 56, 58, 89, 90, 91 und 137 zitiert.)

[Mat11] Gabriel N. Matni. Operating system level trace analysis for automated problem identification. *The Open Cybernetics & Systemics Journal*, 5(1):45–52, jun 2011. doi: 10.2174/1874110x01105010045. (Auf den Seiten 4, 50 und 51 zitiert.)

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976. doi: 10.1109/TSE.1976.233837. (Auf Seite 2 zitiert.)

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991. doi: 10.1145/103727.103729. (Auf Seite 26 zitiert.)

[Mic13] Michael Kerrisk. Lca: The trinity fuzz tester. https://lwn.net/Articles/536173/, February 2013. Zugegriffen: 13.11.2020. (Auf Seite 57 zitiert.)

[MM14] Paul W. McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 279–290, New York, NY, USA, 2014. ACM Press. doi: 10.1145/2597008.2597149. (Auf Seite 41 zitiert.)

[MNNW14] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley, Upper Saddle River, NJ, 2. ed. edition, 2014. (Auf den Seiten 1, 4, 31, 35, 36, 49, 51, 60, 62 und 114 zitiert.)

[Moo65] Gordon E Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, pages 114–117, 1965. (Auf Seite 1 zitiert.)

[MR18] Katharina Morik and Wolfgang (Editors) Rhode. Technical report for collaborative research center sfb 876 - graduate school. Technical Report 6, TU Dortmund University, December 2018. Abschnitt Subproject A1 – Alexander Lochmann. (Auf Seite 81 zitiert.)

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* Wiley Publishing, New York, NY, USA, 3rd edition, 2011. (Auf Seite 56 zitiert.)

[Nat07] National Aeronautics and Space Administration. Powerless. *System Failure Case Studies*, 1(10), December 2007. (Auf Seite 2 zitiert.)

[NC16] Vegard Nossum and Quentin Casasnovas. Filesystem fuzzing with American Fuzzy Lop. In *Proceedings of the Linux Storage and Filesystems Conference (VAULT).* USENIX Association, 2016. (Auf Seite 57 zitiert.)

[Nei16] Neil Brown. Sparse: a look under the hood. https://lwn.net/Articles/689907/, June 2016. Zugegriffen: 19.05.2020. (Auf den Seiten 46 und 47 zitiert.)

[NHD+20] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020. (Auf Seite 42 zitiert.)

[NM92] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992. doi: 10.1145/130616.130623. (Auf Seite 14 zitiert.)

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM Press. doi: 10.1145/1250734.1250746. (Auf den Seiten 4, 48, 50 und 51 zitiert.)

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer Conference*, pages 247–256. USENIX, June 1990. (Auf Seite 28 zitiert.)

[OW02] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 55–64, New York, NY, USA, 2002. ACM Press. doi: 10.1145/566172.566181. (Auf Seite 3 zitiert.)

[OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 86–96, New York, NY, USA, 2004. ACM Press. doi: 10.1145/1007512.1007524. (Auf Seite 3 zitiert.)

[PAJ18] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium*, pages 729–743, Baltimore, MD, August 2018. USENIX Association. (Auf den Seiten 55 und 90 zitiert.)

[Par16] Byungchul Park. Enhancing lockdep with crossrelease, December 2016. (Auf Seite 49 zitiert.)

[Pau07a] Paul McKenney. What is RCU, fundamentally? https://lwn.net/Articles/262464/, December 2007. Zugegriffen: 14.04.2021. (Auf den Seiten 21 und 22 zitiert.)

[Pau07b] Paul McKenney. What is RCU? part 2: Usage. https://lwn.net/Articles/263130/, December 2007. Zugegriffen: 14.04.2021. (Auf Seite 22 zitiert.)

[Pau08] Paul McKenney. RCU part 3: the RCU api. https://lwn.net/Articles/264090/, January 2008. Zugegriffen: 15.04.2021. (Auf Seite 22 zitiert.)

[PDEH15] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, New York, NY, USA, 2015. ACM Press. doi: 10.1145/2797022.2797042. (Auf den Seiten 27 und 28 zitiert.)

[Pet83] Gary L. Peterson. A new solution to lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. Syst.*, 5(1):56–65, January 1983. doi: 10.1145/357195.357199. (Auf Seite 25 zitiert.)

[Pro20] The FreeBSD German Documentation Project. FreeBSD Handbuch. https://www.freebsd.org/doc/de/books/handbook/, 11 2020. Version: 45e8327411, abgerufen am 17.12.2020. (Auf Seite 121 zitiert.)

[Pro21] The FreeBSD Documentation Project. Freebsd architecture handbook. https://docs.freebsd.org/en/books/arch-handbook/, 01 2021. Version: eab1c5d1f6, abgerufen am 29.04.2021. (Auf Seite 54 zitiert.)

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, sep 1994. doi: 10.1145/186025.186041. (Auf Seite 43 zitiert.)

[Ree04] Kenneth A. Reek. Design patterns for semaphores. *SIGCSE Bull.*, 36(1):320–324, March 2004. doi: 10.1145/1028174.971412. (Auf Seite 15 zitiert.)

[RST+15] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: Formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 38–53, New York, NY, USA, 2015. ACM Press. doi: 10.1145/2815400.2815411. (Auf den Seiten 4 und 42 zitiert.)

[Rus] Rusty Russell. Kernel hacking guides: Unreliable guide to locking. https://www.kernel.org/doc/html/v5.5/kernel-hacking/locking.html. Accessed: 2020-03-30. (Auf Seite 32 zitiert.)

[SAG+17] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium*, pages 167–182, Vancouver, BC, August 2017. USENIX Association. (Auf Seite 58 zitiert.)

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, nov 1997. doi: 10.1145/265924.265927. (Auf den Seiten 4, 41, 47, 51, 59, 60, 65, 67 und 82 zitiert.)

[SGG10] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts Essentials*. Wiley Publishing, 2010. (Auf den Seiten 15, 26 und 29 zitiert.)

[Sha19] Eugene Shatokhin. KernelStrider. https://github.com/euspectre/kernel-strider, 2019. (Auf den Seiten 4, 50 und 51 zitiert.)

[SHD+15] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 245–255, Piscataway, NJ, USA, September 2015. IEEE Press. doi: 10.1109/EDCC.2015.28. (Auf den Seiten 9, 55 und 72 zitiert.)

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM Press. doi: 10.1145/1791194.1791203. (Auf den Seiten 48 und 50 zitiert.)

[SNW08] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux Applications*. Network Theory Ltd., 2008. (Auf den Seiten 48 und 50 zitiert.)

[Som16] Ian Sommerville. *Software Engineering, Global Edition*. Pearson Education Limited, Harlow, UNITED KINGDOM, 2016. (Auf den Seiten 3 und 23 zitiert.)

[SP94] Wolfgang Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, Upper Saddle River, NJ, USA, 1994. (Auf Seite 20 zitiert.)

[SPIV12] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 110–114, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. (Auf den Seiten 48, 50 und 54 zitiert.)

[SSSS00] F. Schon, W. Schroder-Preikschat, O. Spinczyk, and U. Spinczyk. On interrupt-transparent synchronization in an embedded object-oriented operating system. In *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000) (Cat. No. PR00607)*, pages 270–277, 2000. doi: 10.1109/ISORC.2000.839540. (Auf Seite 20 zitiert.)

[Sta82] Eugene W. Stark. Semaphore primitives and starvation-free mutual exclusion. *J. ACM*, 29(4):1049–1072, October 1982. doi: 10.1145/322344.322352. (Auf den Seiten 25 und 26 zitiert.)

[Sta11] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 7th edition, 2011. (Auf den Seiten 12, 14, 15, 16, 17, 18, 19, 20, 25, 26, 28 und 29 zitiert.)

[SWF+19] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. Industry practice of coverage-guided enterprise Linux kernel fuzzing. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 986–995, New York, NY, USA, 2019. ACM Press. doi: 10.1145/3338906.3340460. (Auf Seite 58 zitiert.)

[TB14a] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th edition, 2014. (Auf den Seiten 11, 12, 13 und 15 zitiert.)

[TB14b] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, USA, 4th edition, 2014. (Auf den Seiten 15, 20, 26 und 29 zitiert.)

[tre21] Treant.js – Treant-js is an SVG based JS library for drawing tree diagrams. https://github.com/fperucic/treant-js, 2021. Zugegriffen: 08.01.2018. (Auf Seite 81 zitiert.)

[TW05] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation*. Prentice-Hall, Inc., USA, 3rd edition edition, 2005. (Auf Seite 13 zitiert.)

[U.S04] U.S.-Canada Power System Outage Task Force. Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations. Report, United States Department of Energy and Canadian Department of Natural Resources, April 2004. (Auf Seite 2 zitiert.)

[VAR+16] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, 2016. doi: 10.1145/2970276.2970337. (Auf den Seiten 4, 44, 45 und 51 zitiert.)

[vECC+99] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. *J-Kernel: A Capability-Based Operating System for Java*, pages 369–393. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. doi: 10.1007/3-540-48749-2_17. (Auf Seite 42 zitiert.)

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY. 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*. ACM Press, 2007. doi: 10.1145/1287624.1287654. (Auf Seite 43 zitiert.)

[Vyu15] Dmitry Vyukov. Syzkaller: An unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2015. Zugegriffen am 13.11.2020. (Auf den Seiten 57, 58, 90 und 137 zitiert.)

[Vyu16] Dmitry Vyukov. kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/v5.4/dev-tools/kcov.html, 2016. Accessed: 2020-08-20. (Auf Seite 89 zitiert.)

[Wit07] Thomas Witkowski. Formal verification of Linux device drivers. mathesis, Technische Universität Dresden, May 2007. (Auf den Seiten 4 und 42 zitiert.)

[XKZK20] M. Xu, S. Kashyap, H. Zhao, and T. Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020. doi: 10.1109/SP40000.2020.00078. (Auf den Seiten 50, 51 und 92 zitiert.)

[Yos07] Hiro Yoshioka. Regression test framework and kernel execution coverage. In *Proceedings of the Linux Symposium*, pages 285–296, 2007. (Auf Seite 57 zitiert.)

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. doi: 10.1145/267580.267590. (Auf Seite 56 zitiert.)