technische universität
dortmund

Untersuchung der Kernsynchronisation in NetBSD mittels LockDoc

Masterarbeit

Merlin Scholz
20. September 2023

Supervisors:
Prof. Dr.-Ing. Peter Ulbrich
Dr.-Ing. Alexander Lochmann

Fakultät für Informatik
Technische Universität Dortmund
http://www.cs.tu-dortmund.de

INHALTSVERZEICHNIS

1

EINLEITUNG

1.1 MOTIVATION

Ein immer größerer Teil des täglichen Lebens beruht heutzutage auf dem Einsatz von Computern. Obwohl den sich auf diesen Computern befindlichen Anwendungen viel Beachtung geschenkt wird, so würden diese ohne das unterliegende Betriebssystem praktisch nicht funktionieren. Diese Differenz in der Evaluierung findet sich dementsprechend auch oft in Praxis wieder: Für Programme werden Unit-Tests, Integration-Tests und andere Methoden zum Überprüfen der Korrektheit genutzt. Diese sind allerdings fast wertlos, wenn das unterliegende Betriebssystem sich nicht korrekt verhält.

Eine häufige Fehlerursache – sowohl in Betriebssystemkernen als auch in gewöhnlichen Anwendungen – ist die fehlerhafte Synchronisation mehrerer parallel stattfindender Aufgaben. Dies ist allerdings insb. bei den Betriebssystemkernen oftmals eine Schwachstelle, da die jeweiligen Betriebssysteme zu einer Zeit geschaffen wurden, zu der noch keine Mehrkernprozessoren existierten. Dementsprechend wurde erst nachträglich versucht, die Unterstützung der Parallelisierung von Prozessen in den bestehenden Code zu integrieren. Das Einarbeiten dieser Modifikationen dauert über Jahrzehnte an und während die Mitwirkenden mit größter Vorsicht arbeiten, so ist nicht garantiert, dass sie nicht auch Fehler machen, insb. Fehler, die für das menschliche Auge schwer zu erkennen sind.

Da es aufgrund der Komplexität oder der reinen Menge an Code oftmals nur schwer möglich ist, diese Fehler manuell zu erkennen, existieren verschiedene Methoden zur automatisierten Fehlersuche. Allerdings ergeben sich durch den Einsatz dieser Techniken auf der Kernel-Ebene durch die Hardware-Nähe eine Anzahl an neuer Probleme, aber auch neue Möglichkeiten im Bezug auf die nutzbaren Analysevektoren.

1.2 ZIELE

Ziel dieser Arbeit ist es, ein bisher kaum erforschtes Betriebssystem auf Fehler bzgl. der Synchronisation hin zu überprüfen. Durch die Wahl eines Betriebssystems, welches saubere Synchronisation verspricht, besteht somit auch die Möglichkeit, neue Erkenntnisse über die Stärken und Schwächen von LockDoc zu gewinnen und etwaige Schwachstellen zu beseitigen.

Neben der Evaluation von LockDoc selbst besteht ein weiteres Ziel der Arbeit daraus, durch dessen Einsatz auf einem weiteren Betriebssystemkern mögliche Fehler, Unklarheiten oder Dokumentationsprobleme in diesem Kern zu finden und ggf. zu beheben.

1.3 GLIEDERUNG

Begonnen wird zu diesem Zweck mit der Auswahl eines geeigneten, bisher nicht durch LockDoc erforschten Betriebssystems. Es folgt Untersuchung dahingehend, welche Synchronisationsmechanismen innerhalb dieses Betriebssystems relevant für die anstehende Analyse sind, gefolgt von einer Begutachtung möglicher zu instrumentierender Subsysteme.

Anschließend geschieht das Portieren sämtlicher für LockDoc benötigter Module in den gewählten Kern, gefolgt von der Modifikation der Sperren- und Speicherverwaltung dahingehend, als dass diese aufgezeichnet werden können. Zudem wird analysiert, wie groß der Arbeitsaufwand ist, LockDoc in ein neues, bisher nicht unterstütztes Betriebssystem zu integrieren.

Nach dem Instrumentieren der jeweils relevanten Datenstrukturen folgt das Auffinden einer geeigneten Methode, dieses Subsystem zu testen – geschehen kann dies über Stresstests, Benchmarks, in das Betriebssystem integrierte Tests oder andere Methoden.

Die so gesammelten Daten werden nachverarbeitet und mit Hilfe von ihnen werden Code-Pfade, die sich hinsichtlich des Lockings auffällig verhalten, manuell auf mögliche Fehler hin inspiziert.

In diesem Schritt geschieht auch eine Analyse dahingehend, welche spezifischen Situationen nicht von LockDoc abgebildet bzw. überprüft werden können. Wenn möglich wird LockDoc so erweitert, als dass diese Sonderfälle unterstützt werden.

2

GRUNDLAGEN

Sperren – oder Locks – sind elementar, um gewährleisten zu können, dass Code ohne miteinander zu interferieren nebenläufig auf einem System ausgeführt werden kann. Während diese Beschreibung auf viele Ausführungsumgebungen zutrifft, so sind Kerne von Betriebssystemen eine wichtige Stelle, an denen Locks genutzt werden, da diese dafür verantwortlich sind, eine Vielzahl an unterschiedlichen Arbeitslasten, oftmals parallel, zu verwalten. Des Weiteren ist der reibungslose Ablauf im Betriebssystemkern dahingehend wichtig, dass dieser die Basis für sämtliche Applikationen bildet. Finden dort Synchronisationsfehler, wie z. B. Race Conditions oder Verklemmungen statt, so kann dies katastrophale Folgen für möglicherweise kritische Programme haben, welche auf diesem Kern laufen.

Um Sperren effizient einzusetzen, muss abgewogen werden, wie grob- oder feingranular diese Sperrenoperationen geschehen sollen. Um dies besser zu veranschaulichen, wird auf Extremfälle in beide Richtungen geblickt.

Der Extremfall der zu groben Synchronisation besteht bspw. aus dem Sperren eines kompletten Subsystems, sobald auf eine seiner Komponenten zugegriffen wird. Dies schadet der Leistung des Systems, es sorgt allerdings für eine relativ hohe Sicherheit bzgl. der Korrektheit der Sperren: Wenn ein ganzes Subsystem gesperrt ist, so besteht eine sehr geringe Wahrscheinlichkeit, dass paralleler Code fälschlicherweise auf die selben Datenstrukturen zugreift. Ein weiterer Vorteil ist der geringe Implementierungsaufwand, da nicht vor dem Zugriff auf gewisse Datenstrukturen überlegt werden muss, ob und wie diese gesperrt werden sollten.

Das Gegenteil besteht aus dem (zu) feingranularen Sperren. In diesem Szenario wird jede einzelne Datenstruktur vor einem Zugriff gesperrt und anschließend wieder entsperrt. Während dies auch zu einer hohen Sicherheit bezüglich der Korrektheit der Synchronisation führt, vorausgesetzt dieses genaue Locking wird konsequent und korrekt implementiert, so ergibt sich wie beim zu groben Locking eine vergleichsweise schlechte Leistung. Diese entsteht durch den Mehraufwand (engl. Overhead) des

Zugriffs auf alle einzelnen Sperren. Zusätzlich sorgt dieser Ansatz für eine erheblich schlechtere Lesbarkeit des Quelltextes, da ein hoher Prozentsatz dessen Zeilen nur aus Code zur Synchronisation der Datenstrukturen besteht.

Im Folgenden wird auf die Ursprünge, die Umsetzung und die sich daraus ergebende Auflösung von Sperren in Betriebssystemkernen eingegangen.

2.1 LOCKING IN BETRIEBSSYSTEMEN

In Betriebssystemkernen existieren verschiedene Ursachen für die Notwendigkeit von Nebenläufigkeit und der sich daraus ergebenden Notwendigkeit für Synchronisation. Einer der intuitivsten Gründe besteht aus der Einführung von Mehrkernprozessoren. Die Entwicklung einiger Betriebssystemkerne bzgl. der Adaption an Mehrkernprozessoren wird im Folgenden beschrieben.

Geschichtlich tendierten (fast alle) Betriebssystemkerne zur Verwendung von sehr groben Methoden zur Synchronisation. Dies rührte daher, dass durch die Einführung von Mehrkernprozessoren mit Unterstützung für Symmetric Multiprocessing (SMP) ein bis an diesen Punkt uneingeschränkter, paralleler Zugriff auf Kernelressourcen verhindert werden musste. Wird bei der Implementierung eines Betriebssystem-Kernels zu so einem kernelweiten Lock gegriffen, so spricht man von einem Big Kernel Lock (BKL). Heutzutage verbreitete Betriebssysteme wie Linux, FreeBSD, OpenBSD oder NetBSD haben mit ihrer Entwicklung begonnen lange bevor die ersten Mehrkernprozessoren entwickelt wurden. So hatte Linux seine Ursprünge bspw. in 1991, während sich Dual-Core CPUs erst nach der Jahrtausendwende für Endnutzer durchgesetzt haben.

Bis zu diesem Trend war es für Kernel nur in einem Szenario zwingend notwendig, Synchronisationsmechanismen einzuführen, diese waren lediglich zum Behandeln von Unterbrechungen (engl. Interrupts) notwendig. Diese zu Implementieren ist im Vergleich zu modernen Szenarien relativ unkompliziert. Auch wenn es verschiedene Level an Unterbrechungen gibt, so konnten diese nicht parallel bearbeitet werden. Die Synchronisation bestand daraus, dass auf die abgeschlossene Behandlung des höher priorisierten Interrupts gewartet wird.

Durch besagte Einführung von Mehrkernprozessoren und dem daraus folgenden Wunsch nach effizienter Nutzung von SMP bestand ein großes Ziel der Betriebssystem-Mitwirkenden aus der nachträglichen Einführung von feingranularen Synchronisationsmechanismen in jedes Subsystem des jeweiligen Kerns.

Bei einem Großteil der Kerne startete diese Optimierung durch die Einführung eines BKL. [Wat07] Dieses sorgte dafür, dass sobald ein Anwendungsprozess (engl. User Mode Process) auf den Kernel zugreifen wollte (wie es bspw. bei einem Systemaufruf, engl. Syscall, passiert), der komplette Kernel für weitere Anfragen von anderen Prozessoren gesperrt wird. Diese mussten warten, bis etwaige vorherige Operationen im Kernel beendet waren.

Unter Linux erfolge die Unterstützung von SMP und damit die Einführung des BKL mit Version 2.0 im Jahr 1996. [BC05] In den darauffolgenden Jahren wurden einzelne Subsysteme nacheinander wieder von der Synchronisation alleinig durch das BKL befreit. Der eigentliche Code, welcher das BKL selbst definiert, wurde jedoch erst zu Version 2.6.39 im Jahr 2011 entfernt. [Cor11]

In FreeBSD wurde SMP-Unterstützung in FreeBSD 3.0 (1997) unter Verwendung eines BKL-Äquivalents namens *Giant Mutex* eingeführt [fre]. Ab Version 5 (2003) wurde mit der noch andauernden Entfernung dieses Giant Locks begonnen, ein Großteil wurde bis Version 6 (2005) entfernt. Ein weiteres größeres Unterfangen zum Entfernen des verbleibenden Codes geschah zu Version 7 (2008). Es ist allerdings für vereinzelte Anwendungszwecke immer noch in FreeBSDs Kernel vorhanden. [Wat07]

Dieser Ansatz des Einführens und wieder Entfernens des BKL ermöglichte es, nach und nach einzelne Datenstrukturen, Methoden und anschließend ganze Subsysteme für parallele Ausführung vorzubereiten. Der Prozess des Entfernens des BKL beinhaltet typischerweise das Einführen von entweder globalen Sperren für eine genau festgelegte Menge an Operationen oder das Hinzufügen von einer oder mehreren Sperren in die zu instanziierenden Datenstrukturen selbst. Dadurch ergibt sich einerseits eine vergleichsweise feingranulare Richtlinie bzgl. des Sperrens von Datenstrukturen, dadurch dass nicht in Verbindung stehende Teile des Kernels keine Sperren teilen mussten, ergab sich aber auch eine Steigerung der Performance für nebenläufige Routinen.

Ein weiterer Vorteil dieser Vorgehensweise besteht darin, dass sich die Mitwirkenden zuerst den besonders häufig aufgerufenen Routinen widmen konnten, statt alle Kernel-Subsysteme gleichzeitig verbessern zu müssen.

Neben diesem offensichtlichen Fall des SMP existieren noch weitere Gründe, aus denen Nebenläufigkeit im Betriebssystemkern stattfinden kann. Diese Situationen finden auch in Uniprozessorsystemen statt, also in Systemen, welche technisch nicht zur echter Parallelität in der Lage sind. [Sta17]

Ein Beispiel besteht daraus, dass in solchen Umgebungen Unterbrechungen eine Möglichkeit für externe Signale wie bspw. Kommunikationen von anderen Systemen oder Timer-Operationen bieten, den aktuell laufenden Code des eigentlichen Systems zu unterbrechen, sodass zuerst das eingehende Signal – der Interrupt – verarbeitet werden kann. [Sta17] Es ist hier notwendig, dass die jeweiligen Routinen zur Unterbrechungsbehandlung (engl. Interrupt Handler) auf keine Datenstrukturen zugreift, welche vom ursprünglichen Kontrollfluss genutzt werden. Um dies zu garantieren, kann eine Kernel-Routine im kritischen Abschnitt ihres Kontrollflusses zeitweise Unterbrechungen deaktivieren.

2.2 SEMANTISCHE LÜCKE

Durch diese bisher beschriebenen Zusammenhänge ergibt sich eine semantische Lücke zwischen zwei wichtigen Konzepten:

MODELL Wie Sperren logisch gesehen eingesetzt werden sollten

REALITÄT Wie Sperren in der Praxis eingesetzt worden sind

Im Folgenden wird auf die Ursachen dieser Differenz eingegangen, anschließend auf mögliche Konsequenzen.

2.2.1 *Ursachen*

Dieses Delta zwischen Modell und Realität kann verschiedene Ursachen haben. Wichtig ist, zwischen bewussten/absichtlichen und unbewussten/unabsichtlichen Entscheidungen zu differenzieren.

NICHT VORSÄTZLICH Die intuitiv erste Fehlerursache ist die des einfachen Fehlers in der Implementierung. Auch Kernel-Mitwirkende mit langjähriger Erfahrung können inkorrekten Code schreiben, dies kann entweder einen einfachen Denkfehler als Ursache haben, es kann sich aber auch um eine Folge von unzureichender, inkorrekter oder nicht auffindbarer Dokumentation handeln (vgl. Abschnitt 3.1).

VORSÄTZLICH Das Gegenteil der unbeabsichtigten Differenzen zwischen Modell und Realität besteht aus vorsätzlich „inkorrekt“ implementiertem Code. Programmierende

würden in so einem Szenario Aussagen treffen wie „Wir können auf diese Sperre verzichten, da keine Nebenläufigkeit geschehen kann“ oder „Das Nichtüberprüfen dieser Sperre ist an dieser Stelle ungefährlich, weil wir als einzige von der Existenz des Objektes wissen“. Dies ist oftmals der Fall bei der Initialisierung von neu allozierten Speicherbereichen.

Dieser Umstand wurde bereits von Lochmann *et. al.* beschrieben:

> Following a „better safe than sorry“ strategy, it would make sense to acquire all locks that seem to be related to the data structure of interest. This might unintentionally limit parallelism and thus lead to performance degradation. [LSBS19]

Als spezifisches Beispiel ist ein Auszug aus NetBSDs `uvm_aobj.c`[1] zu nennen:

```
/*
 * allocate hash/array if necessary
 *
 * note: in the KERNSWAP case no need to worry about locking since
 * we are still booting we should be the only thread around.
 */
```

Ein weiterer Fall, in welchem oftmals auf Sperren verzichtet wird, ist der Lesezugriff auf Datenstrukturen mit Wortgröße, da dieser je nach Kontext als atomar angesehen werden kann. Lochmann *et. al.* haben dies vermehrt im Bezug auf den Linux Kern festgestellt. [Loc21]

Wird aus solchen Gründen auf den Einsatz von Sperren verzichtet, ist es allerdings fast unmöglich das Wort „inkorrekt“ zu definieren. So stellt sich die Frage ob Code, der zwar immer fehlerfrei den Zweck erfüllt für den er implementiert wurde, dabei aber gewisse Aspekte der Dokumentation missachtet, inkorrekt ist.

2.2.2 *Dokumentation*

Das Implementieren von neuen Kernel-Komponenten oder auch nur das Bearbeiten solcher, erfordert Wissen darüber, welche Sperren vor dem Zugriff auf welche Datenstrukturen eingesetzt werden müssen. Um dies zu erfahren, wird sich i. d. R. an

[1] https://nxr.netbsd.org/xref/src/sys/uvm/uvm_aobj.c?r=1.157#454

vorhandene Dokumentation gewandt, diese sollte wenn möglich eine Art „Single Source of Truth“ bzgl. des Lockings darstellen.

Dies ist allerdings nicht immer realitätsnah: Wie allgemeine Dokumentation von Kern-internen Funktionen ist auch die Dokumentation zur korrekten Synchronisation und insb. wo welche Datenstrukturen wie und warum gesperrt werden müssen, oftmals unzureichend. Dies startet mit den verschiedenen Arten der Dokumentation. So begegnen Entwickelnden, welche bspw. neuen Dateisystem-Treiber implementieren, Dokumentationen in Form von Prosa Texten in den Handbuchseiten (NetBSD 10.99.5, `man 9 vnode`):

> The vnode lock is acquired by calling vn_lock(9) and released by calling VOP_UNLOCK(9). The reason for this asymmetry is that vn_lock(9) is a wrapper for VOP_LOCK(9) with extra checks, while the unlocking step usually does not need additional checks and thus has no wrapper.

Wären diese Handbuchseiten die einzige Quelle für Hinweise bzgl. des Lockings oder der Implementierung im Allgemeinen, so gäbe es eine zentrale Anlaufstelle für Dokumentation. Allerdings finden sich hilfreiche Kommentare auch im Quellcode direkt in Form von Prosa Texten[2], in Quellcode Kommentaren in Form von auskommentierten Code-Beispielen[3] oder sogar in eingebetteten ASCII-Diagrammen[4]. Des Weiteren kommt es vor, dass gewisse logische Zusammenhänge nur in Kernel-Mailinglisten, IRC-Chats oder im schlimmsten Falle gar nicht dokumentiert sind.

2.2.3 *Konsequenzen*

Ein häufiges Beispiel von Konsequenzen dieser unzureichenden Dokumentation lässt sich in den Aufrufkonvention (engl. Calling Conventions) von einzelnen Kernelfunktionen auffinden: Existiert so bspw. eine Funktion, welche auf einer gewissen Datenstruktur agiert, welche durch einen Mutex synchronisiert werden sollte, so ist nicht intuitiv klar, ob die aufrufende Funktion (engl. Caller) oder die aufgerufene Funktion (engl. Callee) dafür verantwortlich ist, diese Sperre entsprechend zu setzen. Ist nun keine

[2] https://nxr.netbsd.org/xref/src/sys/kern/vfs_vnode.c?r=1.149#134
[3] https://nxr.netbsd.org/xref/src/sys/kern/sys_futex.c?r=1.19#43
[4] https://nxr.netbsd.org/xref/src/sys/kern/vfs_cache.c?r=1.155#110

Dokumentation an den erwarteten Stellen zu finden (oder schlichtweg nicht vorhanden), so bleibt es Entwickelnden nur übrig, sich bestmöglich an existierende Funktionen anzupassen oder den Quellcode der aufgerufenen Funktion genauer zu analysieren.

Eine weitere Problematik entsteht, wenn zwar Dokumentation vorhanden ist, diese aber von der eigentlichen Implementierung abweicht. In diesem Kontext kann existierender Code als eine zusätzliche Form von Dokumentation angesehen werden. Diese Differenz zwischen Code und Dokumentation kann unterschiedliche Ursachen haben, dazu gehören u. a. unterschiedliche Autoren der jeweiligen Elementen, die Anpassung des Codes aber nicht der Dokumentation (oder umgekehrt) über die Jahre oder auch etwaige Sprachbarrieren der Entwickelnden. Es stellt sich auch hier wieder die Frage danach, welche Interpretation nun die „korrekte“ ist.

2.3 ANALYSE DES LOCKINGS

Die Analyse von Sperren in Betriebssystemkernen war schon mehrfach Thema von diversen wissenschaftlichen Arbeiten und Ausarbeitungen und ist bis heute ein aktives Forschungsgebiet. An dieser Stelle wird ein Überblick über unterschiedliche Analysemethoden geschaffen und anschließend der LockDoc-Ansatz, den diese Arbeit als Basis nutzt, detaillierter betrachtet.

Allgemein lassen sich unterschiedliche Methoden zur Überprüfung von Programmcode auf dessen Nutzung von Sperren in die vier in von Engler *et. al.* aufgestellten Kategorien unterteilen: Dynamische Analyse, statische Analyse, Post-Mortem Analyse und Model Checking.

Jede dieser Methoden hat ihre eigenen Stärken und Schwächen:

2.3.1 *Dynamische Analyse*

Die dynamische Analyse (auch On-the-fly-Analyse genannt) geschieht durch die vorherige Modifikation des Quelltextes des zu analysierenden Programms. Anschließend wird dieses Programm mit dem Ziel ausgeführt, möglichst viele Code-Pfade abzudecken. Zur Laufzeit werden innerhalb des Programmes relevante Informationen gesammelt, u. a. darüber, welche Sperren wann gesperrt sind und auf welche lokalen oder globalen Variablen zugegriffen wird. Je nach Implementierung wird auch aufgezeichnet, auf welche Speicherbereiche zugegriffen wird.

Dieser Ansatz besitzt einige wichtige Vorteile, dazu gehört, dass nur tatsächlich ausführbare Code-Pfade analysiert werden. [EA03] Dies mag zwar offensichtlich klingen, jedoch besteht bei anderen Analyse-Methoden die Chance, dass unerreichbare Pfade analysiert werden und somit false Positives bzgl. möglicher Synchronisationsfehler erzeugt werden.

Eine weitere Stärke der dynamischen Analyse ist, dass keine Abstraktionsschicht genutzt wird, durch welche non-triviale Operationen wie bspw. Aufrufe von Funktionsaliasen verloren gehen können. Außerdem können so diese Daten zur Laufzeit analysiert werden, was einen zusätzlichen Speicheraufwand durch permanentes Aufzeichnen der jeweiligen Speicher- und Sperrenzugriffe eliminiert. Durch diese beiden Tatsachen ist es möglich, eine größere Menge an Daten in die eigentliche Analyse miteinzubeziehen insb. im Vergleich zu anderen Methoden (vgl. Unterabschnitt 2.3.2), bei denen die Men-

ge an Daten aufgrund von beschränkten Speicherkapazitäten evtl. reduziert werden muss. [HM94]

Allerdings birgt die dynamische Analyse auch Gefahren. So ergibt sich durch die Analyse zur Laufzeit ein hoher rechnerischer Mehraufwand (wenn auch nur ein geringer zusätzlicher Speicher-Aufwand). Im schlimmsten Falle kann dieser Umstand zur Folge haben, dass die Ausführung des Programmes soweit beeinflusst wird, dass gewisse Fehler nicht auftreten und somit nicht gemessen werden können. Außerdem kann dies die Ausführung (und damit die Analyse) von zeitsensitiven Programmen komplett verhindern. [EA03]

Als letzter wichtiger Nachteil ist nennenswert, dass zum Zeitpunkt der Analyse einer spezifischen Operation nur die Daten von bisherigen Operationen zur Verfügung stehen.[5] Dadurch ist es nur eingeschränkt (oder gar nicht) möglich, Sperrenoperationen in Beziehung zu weiteren Zugriffen auf Datenstrukturen zu setzen.

Eine der wohl wichtigsten Überlegungen vor dem Einsatz von dynamischer Analyse ist die, wie sichergestellt werden kann, dass möglichst viele Code-Pfade ausgeführt werden. Es wurde bereits als Vorteil genannt, dass nur tatsächlich ausführbare Pfade analysiert werden. Das Inverse stellt allerdings auch ein Problem dar: Wie kann garantiert werden, dass *möglichst viele* dieser ausführbaren Pfade auch wirklich der Analyse unterzogen werden? Hierbei ist nicht nur die reine Code-Überdeckung relevant, sondern auch die Anzahl der unterschiedlichen Aufrufe einer beliebigen Funktion, da innerhalb dieser Aufrufe oftmals unterschiedliche Sperren gesetzt werden. Dieses nicht-uniforme Verhalten der bereits gesetzten Sperren stellt eine der wichtigsten in dieser Arbeit zu analysieren Fehlerursachen dar. Erschwert wird die dynamische Analyse zusätzlich durch die Tatsache, dass augenscheinlich simple Optionen, wie das Aktivieren oder Deaktivieren von Debug-Ausgaben, die Nutzung unterschiedlicher Übersetzer oder sogar die aktuelle Uhrzeit dazu führen können, dass gewisse Fehler entweder auftreten oder eben nicht auftreten können. [EA03]

[5] Auch dies ist nicht zwingend gegeben, da nicht alle dynamischen Ansätze Informationen über vergangene Operationen permanent aufzeichnen.

2.3.2 *Post-Mortem Analyse*

Die Post-Mortem Analyse trennt die dynamische Analyse in zwei Teile auf: Der Quelltext muss immer noch instrumentiert werden, um das Aufzeichnen von Sperrenoperationen oder relevanten Speicherzugriffen zu ermöglichen. Die eigentliche Analyse erfolgt jedoch nicht zur Laufzeit des Programmes sondern separat von diesem. Dieses Auslagern ermöglicht ein einfaches Wiederverwenden der Analyse-Werkzeuge, da diese nicht mehr direkt im Programm integriert sind, siehe LockDoc (vgl. Abschnitt 2.4).

Durch das Aufteilen von Ausführung und Analyse entfällt jedoch ein Vorteil der dynamischen Analyse: Dadurch, dass jegliche Operationen aufgezeichnet werden müssen, ergibt sich ein erheblicher Speicheraufwand, dadurch wird es oftmals nötig, die Menge der Daten zwecks Aufzeichnung zu reduzieren. Das könnte die Abdeckung oder die Auflösung senken. Diese Problematik wurde bereits von Helmbold *et. al.* beschrieben:

> This is somewhat different from the problem generally addressed in post mortem trace analysis where an attempt is made to determine orderings between all blocks (without regard to exactly which shared memory locations were accessed, as space limitations generally prevent this information from being saved for post mortem analysis). [HM94]

Diese Einschränkung stellt mit zunehmend größeren Speicherkapazitäten der letzten Jahrzehnte allerdings kein großes Hindernis mehr da, wie in dieser Arbeit gezeigt wird.

Außerdem wird durch die Post-Mortem Analyse der genannte letzte Nachteil der dynamischen Analyse eliminiert: Dadurch, dass zum Zeitpunkt der eigentlichen Analyse bereits sämtliche Sperrenoperationen und Datenzugriffe geschehen sind, ist eine heuristische Analyse über die Menge aller – auch zeitlich in der relativen Zukunft liegenden – Sperren möglich.

2.3.3 *Statische Analyse*

Der dynamischen Analyse steht die statische Analyse gegenüber. Bei dieser Methode wird das zu untersuchende Programm nicht ausgeführt, stattdessen wird der Code selbst analysiert. Offensichtlicher Vorteil dieses Ansatzes ist, dass der jeweilige Code nicht bearbeitet werden muss und somit nicht von der Analyse selbst beeinflusst wird. Außerdem können so die Code-Pfade getestet werden, welche nur unter ungewöhnlichen

Bedingungen ausgeführt werden. Dadurch wird der größte Nachteil der dynamischen Analyse eliminiert.

Um statische Analyse effizient nutzen zu können muss jedoch eine gewisse Anzahl an Annotationen zu dem zu untersuchenden Quellcode hinzugefügt werden. Flanagan *et. al.* nennen in [FF00] eine Größenordnung von etwa 20 Annotationen pro 1.000 Zeilen Quellcode. Da dieser Aufwand mit der Anzahl der Zeilen Quellcode wächst, ergibt sich somit für Betriebssystemkerne ein weiteres Problem: In [EA03] wird von Engler *et. al.* als grober Maßstab ein Arbeitsaufwand von etwa 2.400 Stunden reiner Annotation eines Betriebssystemkerns genannt. Da diese Arbeit im Jahr 2003 veröffentlicht wurde und Betriebssystemkerne seither stark angewachsen sind, ist auch ein wesentlich größerer Arbeitsaufwand bzgl. der Annotation notwendig.

Eine Limitierung die insb. im Bereich der Betriebssystemkerne sehr relevant ist, ist der Einsatz von Zeigern in C: Da ein Werkzeug zur statischen Analyse keine Informationen über den Speicherinhalt generiert, können direkt speicherabhängige Operationen nicht rein statisch Analysiert werden. [LSBS19] Das Themengebiet der Zeigeranalyse stellt weiterhin ein aktives Forschungsgebiet dar: [HM94, SB+15] Engler *et. al.* verallgemeinern diese Aussage in [EM04] dahingehend, dass Werkzeuge zur statischen Analyse allgemein Schwierigkeiten bzgl. der Überprüfung von dynamisch allozierten Objekten besitzen.

2.3.4 *Model Checking*

Als weiterer Ansatz zur Analyse von Code bleibt das Model Checking. Bei diesem wird die zu untersuchende Codebasis in ein vereinfachtes aber im Bezug auf Sperren und Parallelität äquivalentes Modell umgewandelt, dieses besteht oftmals aus einem Zustandsdiagramm. [GCP99] Ähnlich der dynamischen Analyse werden in diesem Modell möglichst viele Codepfade „ausgeführt“ und dabei auf etwaige Fehler untersucht. Der wichtigste Unterschied im Vergleich zu anderen erläuterten Methoden besteht aus der Notwendigkeit des Übersetzens des Programmes in ein adäquates Modell. Dies führt zu einem erhöhten Arbeitsaufwand entweder zur einmaligen Implementierung eines automatisierten Übersetzers oder zur manuellen Übersetzung jeder zur prüfenden Anwendung. Letzteres benötigt im Idealfall Mitwirkende des ursprünglichen Projektes, da es ein tiefgreifendes Verständnis der Codebasis voraussetzt. [EM04]

Bei diesem Prozess besteht zudem die Gefahr, dass das so entstehende Modell von der ursprünglichen Codebasis abweicht. Es ist zudem nicht immer möglich, das

originale System perfekt abzubilden, so muss oftmals auf manche Aspekte wie Caches, I/O-Systeme oder Kommunikation mit externen Systemen verzichtet werden. [EM04]

Das Model Checking bietet allerdings auch Vorteile gegenüber anderen Methoden. Insbesondere im Gegensatz zur statischen Analyse ist es besser dazu geeignet, dynamisch allozierten Speicher in die Analyse miteinzubeziehen. Außerdem sinkt durch die Nutzung von Model Checking der Anteil an false Positives, da für jeden möglichen Fehler ein kompletter Stacktrace zurückgegeben werden kann – somit ist problematischer Code bewiesen erreichbar. [EM04]

2.4 SPERRENANALYSE VIA LOCKDOC

Wie bereits in Kapitel 1 beschrieben, ist das Ziel dieser Arbeit die Analyse des NetBSD Kernel-Codes auf etwaige Fehler bzgl. des Lockings.

Für diese Suche nach Programmierfehlern gibt es verschiedenste Ansätze, einige wurden in den vorangegangenen Abschnitten erläutert. Es wird jedoch schnell ersichtlich, dass der Versuch einer statischen Analyse schnell an seine Grenzen trifft. Dies hat verschiedene Ursachen, die wichtigsten jedoch sind auf die pure Größe der Code-Basis und unvorhersehbare Einflüsse durch externe Umstände (wie das interne Verhalten von Hardware bei der Analyse von Treibern) zurückzuführen. Des Weiteren hat die statische Analyse im Allgemeinen gewisse Limitierungen, welche es nicht möglich machen, jeden Code-Pfad auszureichen zu analysieren.

Aus diesen Gründen wird im in [LSBS19] vorgestellten LockDoc-Ansatz statt statischer Analyse zur experimentbasierten Analyse (insb. zur Post-Mortem Analyse) gegriffen. Bei dieser wird „das laufende System beobachtet“, was viele neue Analysevektoren ermöglicht.

2.4.1 *Vorgehensweise*

LockDoc basiert auf der Annahme, dass Code *meistens* korrekt implementiert ist und funktioniert:

> Der Großteil des zu untersuchenden Programmcodes arbeitet korrekt. Fehler sind hingegen rar. [Loc21]

Diese Behauptung kann dadurch belegt werden, dass bei ähnlicher Dichte von Sperrenoperationen pro Code-Pfad ein Großteil der Sperrenoperationen in häufig ausgeführtem Code stattfindet. Die zu analysierenden Betriebssysteme sind oftmals weit verbreitet und pro Instanz des jeweiligen Kernels werden die häufig ausgeführten Code-Pfade sehr regelmäßig genutzt: Daher kann schlussfolgernd davon ausgegangen werden, dass etwaige Race Conditions, Verklemmungen o. Ä. bereits vermehrt aufgetreten sind und somit bereits den jeweiligen Mitwirkenden gemeldet und von diesen beseitigt wurden. Die Hypothese lässt sich auch invers dadurch belegen, dass ein Betriebssystemkern, dessen Quellcode diverse Probleme in häufig genutzten Operationen beinhaltet, instabil ist und somit nicht weit verbreitet ist.

Diese Annahme wird sich in LockDoc zunutze gemacht: Für eine beliebige Datenstruktur (oder Teile dieser) wird davon ausgegangen, dass vor einem Zugriff auf diese die entsprechenden Sperren *meistens* korrekt eingesetzt wurden. Da diese Datenmenge in einer Post-Mortem Analyse komplett zur Verfügung steht, werden heuristische Mittel genutzt, um Hypothesen darüber aufzustellen, welche Menge (und ggf. Reihenfolge) an Sperren vor dem Lesen oder Schreiben einer Variable eingesetzt werden muss.

2.4.2 *Funktionsweise*

Das in Abbildung 2.4.1 dargestellte Flussdiagramm stellt eine Übersicht über den Ablauf der LockDoc-Experimente dar. Im Folgenden wird dieser genauer erläutert.

2.4.3 *Monitoring/Tracing*

Startpunkt der LockDoc Analyse bildet das zu untersuchende Betriebssystem bzw. dessen Kern. Dieses wird so vorbereitet, als dass es neben den zu untersuchenden Subsystemen eine möglichst geringe Anzahl an Modulen enthält, um mögliche Interferenzen oder Overhead gering zu halten. Innerhalb dieses Betriebssystems wird anschließend eine Arbeitslast ausgeführt, deren Ziel es ist, eine große Anzahl an unterschiedlichen Code-Pfaden der entsprechenden Subsysteme auszuführen und somit für Speicher- und Sperrenzugriffe zu sorgen.

Um diese Zugriffe zu erzeugen bedarf es einer Arbeitslast, welche eine möglichst breite Menge an unterschiedlichen Sperrenoperationen ausführt. Idealerweise sorgt diese für einen einzelnen Aufruf pro zu testender Funktion, um alle Funktionen gleich zu gewichten. In der Realität ist dies allerdings nicht möglich, da zwischen den jeweiligen Funktionen oft Abhängigkeiten existieren, welche das direkte und unabhängige Ausführen eines einzigen Code-Blocks unmöglich machen. Zu diesem Zweck eignen sich Test Suites, welche darauf abzielen eben diese Kernel-Komponenten selbst zu evaluieren.

Lochmann *et. al.* nutzen hierfür das Linux Test Project. Das LTP wurde von Mitwirkenden des Linux Kernels geschaffen, um den Linux Kernel auf Fehler oder Regressionen hin prüfen zu können. Intern besteht das LTP aus sogenannten Test Suites, welche wiederum eine Menge an Tests enthalten.

Der Kern des Betriebssystems ist so modifiziert, als dass dieser jegliche Sperrenoperationen aufzeichnen kann und relevante Speicherbereiche zur späteren Analyse markieren

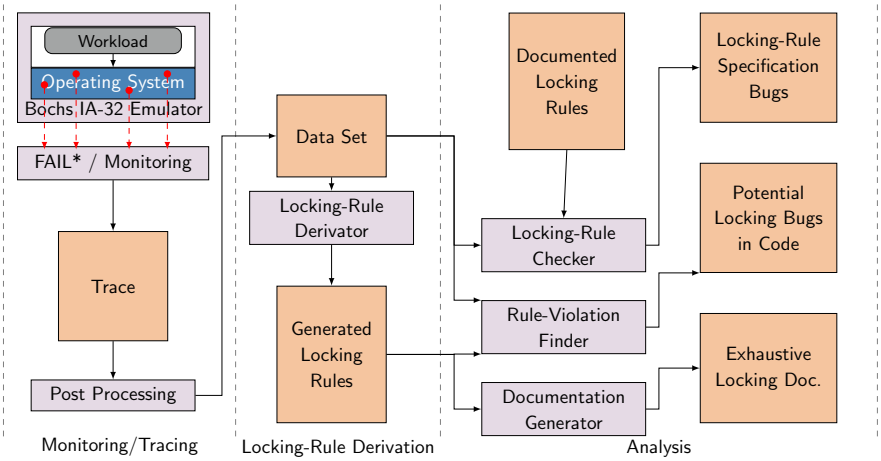

Abbildung 2.4.1: Funktionsweise der LockDoc Experimente, aus [Loc21]

kann. Das Modifizieren des Kerns dahingehend diese Allokationen aufzuzeichnen wird als Instrumentieren bezeichnet. Die eigentliche technische Implementierung dieser Aufzeichnung basiert auf dem FAIL*-Emulator. Dieser ist ursprünglich ein Projekt zur Fault Injection, welches u. a. den Bochs-Emulator nutzt um ein laufendes, emuliertes System zu beeinflussen und zu beobachten. [SHD+15] Dieser wurde durch ein neues *Experiment* dahingehen erweitert, dass er

1. Konsolen-Nachrichten mit der VM[6] austauschen kann.
2. Speicherzugriffe in zu beobachtenden Bereichen aufzeichnen kann.
3. Ereignisse, welche von der VM übertragen werden, lesen und verarbeiten kann.

Ersteres, der Austausch von Konsolen-Nachrichten über die emulierte serielle Schnittstelle dient lediglich dem einfacheren Debuggen der LockDoc-Experimente.

Das Aufzeichnen der Speicherzugriffe geschieht, sobald der emulierte Kernel auf einen Speicherbereich zugreift (lesend und schreibend), welcher zuvor vom Betriebssystemkern als relevant markiert wurde.

[6] An dieser und folgenden Stellen wird der Begriff VM genutzt, aus technischer Sicht handelt es sich jedoch nicht um eine VM im herkömmlichen Sinne, da jegliche Operationen emuliert statt virtualisiert werden.

Sämtliche Speicher- und Sperren-Zugriffe werden von FAIL* zur Weiterverarbeitung in CSV-Dateien persistiert.

Ist die Ausführung dieser Arbeitslast abgeschlossen, so folgt das Post-Processing. In diesem Schritt werden die gesammelten Daten über Sperren- und Speicher-Zugriffe in Verbindung mit dem Debug-Images des Kerns so weiterverarbeitet, dass für jeden Speicherzugriff eine Menge an sogenannten Transaktionen generiert wird. Diese Transaktionen beinhalten neben Metadaten wie Start- und Endzeitpunkt Informationen darüber, welche Sperren in welcher Reihenfolge zum Zeitpunkt des Speicherzugriffes geholt wurden. Diese Transaktionen starten mit dem Holen einer spezifischen Sperre und enden mit dem Freigeben eben dieser. Dazwischen enthalten sie alle weiteren Sperrenoperationen, die zu dieser Zeit stattfinden, dementsprechend können sie verschachtelt sein.

Zudem finden zum Zeitpunkt der Nachverarbeitung weitere Schritte statt, welche das anschließende Weiterverarbeiten und die spätere manuelle Analyse erleichtern. Zum einen werden die Datensätze von Einträgen bereinigt, die zuvor durch manuell erstellte Filterlisten (engl. Blacklists) als irrelevant markiert wurden, da diese sonst die Ergebnisse beeinflussen könnten. Zu weiteren Schritten der Nachverarbeitung gehört aber auch das Assoziieren von Speicher- und Sperren-Zugriffen mit den zugehörigen Code-Zeilen, sowie das Assoziieren von Speicheradressen mit den zugehörigen Variablen-Namen.

Die durch die Nachverarbeitung entstehenden Datensätze werden zur effizienteren Analyse in eine relationale Datenbank eingepflegt.

2.4.4 *Locking-Rule Derivation*

Wird nun über LockDoc (oder eine andere Methode zur Datengewinnung) ein Datensatz gewonnen, so muss vor dem Aufzählen möglicher Sperrenverstöße zuerst festgelegt werden, was als ein Verstoß gilt. Dafür wird vorausgesetzt, dass für eine Datenstruktur (oder einen Teil dieser) bekannt ist, welche Sperren-Menge (engl. Lock Set) bzw. Sperren-Regel die „richtige“ ist. Es ist an dieser Stelle schwer zu definieren, was für eine Datenstruktur als „richtige“ Sperren-Regel gilt. Es kann bspw. nicht von der Dokumentation als Single Source of Truth ausgegangen werden, da diese auch Fehler enthalten kann (vgl. Unterabschnitt 2.2.2).

2.4.4.1 *Hypothesengenerierung*

Da der Grundgedanke der ganzen Methodik daraus besteht, dass Code *meistens* richtig funktioniert (und er ansonsten nicht genutzt werden würde, vgl. Unterabschnitt 2.4.1), wird sich diese Tatsache zunutze gemacht. Der sogenannte *Hypothesizer* stellt aus den aufgezeichneten Sperren- und Speicher-Zugriffen für jede Datenstruktur eine detaillierte Liste an Hypothesen darüber auf, welche Sperren-Regel die richtige ist. Es konkurrieren vier Algorithmen zur Auswahl einer dieser Hypothesen, es folgt eine Erläuterung dieser.

LOCKSET Der LockSet Algorithmus, vorgestellt von Savage *et. al.*, ist ein relativ trivialer Algorithmus zur Auswertung der genannten Datenmengen. Für jede (beobachtete) Variable v wird eine Menge von Sperrenkandidaten $C(v)$ gebildet, welche anfangs sämtliche bekannten Sperren beinhaltet. Wird nun auf v zugegriffen, so wird $C(v)$ durch $C(v) \cap H(v)$ ersetzt, wobei $H(v)$ die Menge der Locks ist, die bei dem aktuellen Zugriff auf v gesperrt sind. Anders ausgedrückt stellt der LockSet Algorithmus Hypothesen auf, welche nur von Sperrenmengen erfüllt werden, welche 100% Abdeckung für die Datenstruktur haben. Dieser Algorithmus ist vergleichsweise strikt, so wurden in [SBN+97] Methoden geschaffen, um gewisse Komplikationen wie Initialisierungsphasen, Read-Only Datenstrukturen und Read-Write-Locks zu umgehen. Ein Problem des LockSet-Algorithmus besteht daraus, dass zwar mögliche Bugs erkannt werden in denen eine Datenstruktur nicht gesperrt wurde, jedoch keine Bugs, in denen nur eine Teilmenge der benötigten Sperren gesetzt wurden. [Loc21]

Es ist anzumerken, dass dieser Algorithmus mit dem Ziel entwickelt wurde, zur Laufzeit einer dynamischen Analyse zu agieren. Aus diesem Grund war es zu seiner Entwicklung nicht möglich eine Abdeckungsrate von $\leqslant 100\%$ zu implementieren, da zum Zeitpunkt der Analyse einer Sperre ursprünglich keine Informationen über darauffolgende Operationen vorhanden waren.

In [Loc21] werden neben LockSet weitere Algorithmen zur Auswahl von Hypothesen entwickelt, welche die Nachteile von LockSet versuchen zu eliminieren. Dabei wird sich das Vorhandensein des kompletten Datensatzes zur Laufzeit der Analyse zunutze gemacht: Für alle beobachteten Sperren-Kombinationen je Datenstruktur wird berechnet, wie hoch dessen Supportrate ist. Die Supportrate ist der relative Anteil an Zugriffen, bei denen die jeweilige Sperren-Kombination (oder eine ihrer Obermengen) geholt wurde.

TOP DOWN Bei dieser Methode wird i. d. R. die Sperren-Kombination mit der höchsten Supportrate genutzt. An dieser Stelle existieren jedoch zwei Ausnahmen: Es wird nie eine leere Menge an Sperren als Hypothese ausgewählt. Da jede Kombination an Sperren eine Obermenge der leeren Menge ist, so hat die leere Menge konstant eine Supportrate von 100%. Die leere Menge ist als Hypothese offensichtlicherweise nicht Wünschenswert, insb. dann nicht, wenn es weitere Hypothesen gibt, welche annähernd, aber nicht vollständig korrekt implementiert wurden. Ist die leere Menge die einzige Menge mit 100% Supportrate, so wird die Hypothese mit der nächsthöchsten Supportrate genutzt. [Loc21]

Die zweite Ausnahme beim Einsatz des Top Down-Algorithmus besteht aus dem Unterschreiten eines im voraus festgelegten Schwellwertes t_{ac}. Existiert (bis auf die leere Menge) keine Hypothese, welche über jenem Schwellwert liegt, so wird keine Hypothese für die jeweilige betrachtete Datenstruktur aufgestellt.

BOTTOM UP Der Bottom Up-Algorithmus agiert sehr ähnlich zum Top Down-Algorithmus. Hier wird jedoch nicht von 100% Supportrate absteigend nach einer Hypothese gesucht, sondern vom Schwellwert t_{ac} aufsteigend. [Loc21]

Ziel dieses Algorithmus ist es, Kombinationen aus mehreren Sperren zu finden, welche jedoch nicht fehlerfrei eingesetzt werden. Solche würden im Top Down-Algorithmus verloren gehen, da eine Hypothese ermittelt werden würde, welche eine höhere Supportrate hat, jedoch (aufgrund von möglichen Bugs) nicht alle relevanten Sperren besitzt.

SHARPEN Dieser Algorithmus startet für eine Datenstruktur mit einer leeren Menge an Sperren. Von diesem Punkt an werden der Menge Sperren hinzugefügt, solange dadurch die Verschlechterung der Supportrate unterhalb eines vorher festgelegten Schwellwertes t_s liegt, es muss also gelten: $1 - \frac{s_r(\text{neu})}{s_r(\text{alt})} \leqslant t_s$ [Loc21]

Neben der Auswahl einer dieser Algorithmen zur Hypothesen-Auswahl existieren noch zwei weitere Parameter, über welche entschieden werden kann.

2.4.4.2 *Write over Read*

Es kann innerhalb einer Transaktion dazu kommen, dass auf eine einzelne Datenstruktur sowohl lesend als auch schreibend zugegriffen wird. Ist dies der Fall, so kann nicht

direkt unterschieden werden, welche der gesetzten Sperren eingesetzt wurden, um vor den lesenden und welche um vor den schreibenden Zugriffen zu schützen. Wird Write over Read (WoR) aktiviert, so wird die beschriebene Situation in der Nachverarbeitung behandelt, als hätte nur ein Schreibzugriff stattgefunden, da für diese i. d. R. strengere Sperren-Regeln gelten. [LSBS19] Als gegenteilige Methode, also das Berücksichtigen beider Zugriffe, existiert NoWoR, auch als „aggregiert" bezeichnet.

2.4.4.3 *Kontext-sensitiv oder -insensitiv*

Zuletzt ergibt sich noch eine Entscheidung darüber, ob die Analyse kontext-sensitiv oder kontext-insensitiv geschehen soll. Der Unterschied zwischen diesen beiden Varianten entsteht, wenn es Sperren gibt, welche in unterschiedlichen Kontexten, also bspw. unterschiedlichen Prozess-IDs oder in der Unterbrechungsbehandlung gesetzt oder freigegeben werden. [Loc21]

Einerseits kann kontext-insensitiv agiert werden: Hierbei werden für Lese- oder Schreibzugriffe alle zuvor geholten Sperren in Betracht gezogen, unabhängig von dem Kontext in dem sie gesperrt wurden. Diese Methode hat zu Folge, dass durch Kontextwechsel, bspw. durch Preemption oder Interrupts, geholte Sperren in der aktuellen Transaktion existieren, welche potentiell keine Relevanz für die aktuelle Datenstruktur haben.

Dem gegenüber steht die kontext-Sensitivität: In dieser Variante werden nur Sperren in die Analyse mit einbezogen, welche innerhalb des Kontextes geholt wurden, in dem der aktuelle Datenzugriff stattfindet. Dies hat den Vorteil, dass es im Gegensatz zum kontext-insensitiven Ansatz weniger Interferenz aus andere Kontexten gibt, allerdings können so auch wichtige Sperren verloren gehen, welche so verwendet werden sollen, dass diese über verschiedene Kontexte gehalten werden müssen.

2.4.5 *Analysis*

In der letzten Phase von LockDoc werden zuvor gewonnenen Sperren-Regeln genutzt, um eigentliche Fehler und Probleme im Betriebssystemcode zu finden. Dabei werden drei Ziele verfolgt: [Loc21]

1. Der Locking-Rule Checker nutzt den in Phase 1 gewonnenen Datensatz in Kombination mit bereits existierender Dokumentation, um zu überprüfen, ob es Code-Pfade gibt, welche nicht mit ihrer Dokumentation übereinstimmen. Hierfür

ist ein manuelles Übertragen der Dokumentation in ein maschinenlesbares Format nötig.

2. Der *Rule-Violation Finder* nutzt zu den generierten Sperren-Regeln den zuvor gewonnen Datensatz, um Code-Pfade zu finden, welche gegen diese aufgestellten Hypothesen verstoßen. Die so gewonnen Ergebnisse können – nach manueller Analyse – dazu genutzt werden etwaige Bugs zu finden.

3. Der *Documentation Generator* kann dazu genutzt werden, um aus den gewonnenen Hypothesen Dokumentation für nicht oder unzureichend dokumentierte Datenstrukturen zu generieren.

3

PROBLEMANALYSE

Während der in Abschnitt 2.4 vorgestellte LockDoc-Ansatz eine vielversprechende Methode zur größtenteils automatisierten Suche nach Synchronisationsproblemen im Kernel darstellt, so ergeben sich durch den Einsatz dieser Methode noch einige Probleme. Im Folgenden werden diese genauer beschrieben.

3.1 DOKUMENTATION

Wie bereits ausführlich in Unterabschnitt 2.2.2 erläutert kann es sich in vielen Situationen als schwer erweisen auf existierende Dokumentation zuzugreifen. Ursachen dafür können das Nichtvorhandensein oder die Nichtauffindbarkeit dieser sein, aber auch interne Widersprüche oder Widersprüche zu existierendem, funktionierendem Code.

Dies stellt auch ein Problem für LockDoc dar. Während es sich zwar dazu eignet, nicht vorhandene Dokumentation zu generieren, so existieren einige Einsatzmöglichkeiten, in denen es von Vorteil ist, eine vorhandene Dokumentation als Quelle zu haben.

Eine Situation, in der zumindest eine grundlegende Dokumentation benötigt wird, ist das Aufstellen der bereits beschriebenen Filterlisten, welche verhindern, dass gewisse Funktionen oder Datentypen in die eigentliche Analyse mit einbezogen werden. Genauer wird dies in Unterabschnitt 6.2.2 beschrieben. Ein weiterer solcher Fall besteht aus dem Nutzen der Dokumentation zur Aufstellung einer Grundwahrheit (engl. Ground Truth), welche zur Entscheidung für eine Strategie zur Hypothesenauswahl genutzt wird. Auf diesen Einsatzzweck wird in Abschnitt 7.3 genauer eingegangen.

Somit ergeben sich für den Einsatz von LockDoc zwei potentielle Probleme: Einerseits kann eine nicht-vorhandene Dokumentation dabei hinderlich sein nötige Filterlisten aufzustellen oder eine Strategie zur Generierung von Hypothesen auszuwählen. Andererseits kann eine inkorrekte Dokumentation zu inkorrekten Filterlisten bzw. Grundwahrheiten führen, was die Ergebnisse allgemein negativ beeinflussen kann.

3.2 AUFFINDEN VON SPERREN

Bei der Allozierung von Speicher in C wird grundsätzlich zwischen zwei Fällen unterschieden:

1. *Statische Speicherallozierung* geschieht einmalig beim Programmstart, in diesem Anwendungsfall bei der Initialisierung des Kernels. Informationen über die Menge an Speicher, die reserviert werden muss, werden zum Zeitpunkt der Übersetzung in die ausführbare Datei eingebettet.

2. *Dynamische Speicherallozierung* findet statt, wenn ein Programm zur Laufzeit zusätzlichen Speicher für Variablen (oder anderweitig) benötigt. Da dies i. d. R. von der Ausführung des eigentlichen Programmcodes und dessen Eingaben abhängig ist, so kann vor der Ausführung nicht bestimmt werden, wann, an welcher Adresse und wofür dieser Speicher alloziert wird.

Die Implementierung von LockDoc erlaubt es, den ersten Fall – die statische Speicherallozierung – direkt dazu zu nutzen, statische, d. h. instanzunabhängige Sperren, automatisiert zu erkennen und Zugriffe auf diese mitzuschneiden. Dies geschieht, indem die zugehörigen Analysewerkzeuge aus der Kernel-Binärdatei auslesen, welche statisch allozierten Variablen mit einem der bekannten, zu analysierenden Arten von Sperre existieren. Daraus wird eine Zuordnung von Speicheradresse zu Sperrenname gewonnen. Wird zum Zeitpunkt der Analyse nun eine Sperrenoperation analysiert, deren Adresse auf diese Weise bekannt geworden ist, so kann für diese Sperrenoperation der Name der Sperre zur weiteren Analyse genutzt werden.

Dynamische Speicherallozierung hingegen wird in LockDoc nur begrenzt unterstützt: Eines der Grundprinzipien von LockDoc ist es, Zugriffe auf beobachtete, dynamisch allozierte Datenstrukturen zu analysieren. Innerhalb dieses Kontextes kommt es oftmals vor, dass eine Datenstruktur eine Sperre beinhaltet, welche den Zugriff auf die Datenstrukturinstanz oder Teile derer, kontrollieren soll. Ein Beispiel eines solchen, eingebetteten Locks findet sich in Listing 3. Dort sind `krwlock_t vi_lock`, `krwlock_t vi_nc_lock` und `krwlock_t vi_nc_listlock` direkt in das Struct eingebettet und befinden sich somit innerhalb seines zugewiesenen Speicherbereichs.

Dies ist allerdings nicht immer der Fall. Einige Datenstrukturen machen Gebrauch von instanzabghängigen Sperren, welche allerdings nur per Zeiger von der Struktur aus referenziert werden. In Listing 4 trifft dies bspw. auf `kmutex_t *v_interlock` zu. Dieses

Mitglied stellt lediglich einen Zeiger auf eine Variable des Typen `kmutex_t` dar, welche sich außerhalb des eigentlichen Structs befindet.

Eine solche Designentscheidung kann verschiedenen Ursachen haben. Zu einer der häufigsten gehört allerdings die durch die Nutzung von Zeigern implementierbare Möglichkeit, dass sich mehrere Datenstrukturen eine Sperre teilen.

Eine Folge dessen ist aber, dass es LockDoc so nicht möglich ist, alle einer Datenstruktur zugehörigen Sperren aufzufinden, insb. da diese oftmals unabhängig von der Datenstruktur selbst alloziert und initialisiert werden. Diese referenzierten Sperren zur Laufzeit zu finden würde Algorithmen benötigen, welche ähnlich zu denen eines Garbage Collectors sind, da beide Systeme mit limitierten Wissen über den eigentlichen Code analysieren müssen, welche Sperren von welchen Zeigern aus referenziert werden und wann diese nicht mehr benötigt werden. Eine zusätzliche Schwierigkeit bei der Umsetzung einer solchen Lösung in LockDoc besteht aus dem stark eingeschränkten Zugriff auf das Gastsystem. Lediglich roher Speicher kann gelesen[1] werden, des Weiteren existiert Zugriff auf ein Debug-Image des Kernels. Außerdem ist eine genauere Analyse in der Nachverarbeitung der Daten nicht möglich, da zur korrekten Beobachtung der Speicherstrukturen diese schon zur Laufzeit der eigentlichen Experimente bekannt sein müssen.

3.3 ARBEITSLAST

LockDoc kann nur Fehler in Code finden, der während des Experiments durchlaufen wird. Daraus folgt, dass die Arbeitslast, welche zu diesem Zeitpunkt ausgeführt wird, ausschlaggebend für den Erfolg der Analyse ist.

Idealerweise deckt diese Arbeitslast so viele Code-Blöcke wie möglich ab, es gibt allerdings nur eine stark begrenzte Anzahl an Programmen, welche darauf abzielen. Als erstes kommen häufig Unit Tests in den Sinn, da diese explizit dafür geschrieben sind, so viele Funktionen wie möglich zu evaluieren. Für den Linux Kernel existiert so bspw. das LTP, ein Programm auf Anwendungsebene, welches sich zum Ziel gesetzt

[1] Der in LockDoc genutzte FAIL*-Emulator kann den Speicher des Gastsystems auch bearbeiten, dies ist an dieser Stelle allerdings nicht von Relevanz, da so die Operationsweise des Kernels beeinflusst werden würde.

hat, eine hohe Anzahl an Systemaufrufen zu testen. An dieser Stelle ist anzumerken, dass das LTP derzeit etwa 35% des Linux VFS Subsystems abdeckt. [LTS20]

Zwei weitere Besonderheiten bzgl. der Arbeitslast sind hier zu beachten:

Sollten als Arbeitslast etwaige Test Suites wie das LTP genutzt werden, so ist es explizit nicht nötig, dass diese Tests auch erfolgreich sind. Lediglich das Durchlaufen der jeweiligen Code-Pfade ist ausreichend für eine höhere Erfolgsquote in der Nachverarbeitung der Daten.

Außerdem ist es dadurch, dass Systemaufrufe betriebssystemabhängig sind, i. d. R. nicht möglich, ein und dasselbe Programm unverändert zum Testen mehrerer Betriebssysteme zu nutzen. Dies führt ggf. zu einem höheren Arbeitsaufwand beim Versuch, die LockDoc Experimente auf neue Betriebssysteme zu portieren.

3.4 KOMPILERINSTRUKTIONEN

Fortgeschrittene in C implementierte Projekte machen oft Gebrauch von Kompilerinstruktionen, um häufig genutzte Routinen oder Abläufe den Mitwirkenden einfach zur Verfügung zu stellen. Dies kann für jegliche Form der dynamischen oder Post-Processing Analyse, damit auch LockDoc, ein Problem darstellen, da solche Instruktionen zum Zeitpunkt der Kompilierung bereits durch den Präprozessor umgewandelt worden sind. Somit finden sich in der dadurch entstehenden Binärdatei keinerlei Informationen darüber, wie der ursprüngliche Code für eine gewisse Routine zuvor aussah. Dies ist insb. bei LockDoc problematisch, da es den Einsatz von Blacklists erschwert, welche dazu dienen, gewisse Funktionen bei der eigentlichen Nachverarbeitung der Daten zu ignorieren.

3.5 NON-TRIVIALE SPERREN-REGELN

In gewissen Implementierungen von Betriebssystemkernen kann es zu Sperren-Regeln kommen, welche zu kompliziert sind, als dass diese durch reine experimentbasierte Analyse genauer auf Fehler geprüft werden können. Dazu gehört bspw. der Fall, wenn zum Zugriff auf eine Datenstruktur mindestens eine von mehreren Sperren geholt werden muss, die Auswahl und Anzahl eben jener Sperren aber den Entwickelnden überlassen wird. Es finden sich in einigen Betriebssystem-Quelltexten Kommentare wie „Beide Sperren zum Schreiben nötig, eine beliebige dieser Sperren zum Lesen“. Während dies für Schreibzugriffe keine größeren Probleme darstellt, so ist es mit

dem aktuellen Stand der LockDoc-Werkzeuge nicht möglich, die freie Wahl von einer von mehreren Sperren korrekt abzubilden. Stattdessen summiert sich (bei korrekter Umsetzung der Sperren-Regel) die Supportrate der Hypothesen für je eine der beiden Sperren auf 100% auf. Diese Limitation ist Lochmann *et. al.* bereits bei dem Einsatz von LockDoc auf FreeBSD aufgefallen. [LS21] Das Erweitern der LockDoc-Werkzeuge um diese Problematik handhaben zu können würde eine vergleichsweise komplizierte Umstrukturierung des Hypothesizers benötigen.

3.6 WEITERE KOMPLIKATIONEN

Es existieren weiterhin einige Komplikationen beim Einsatz von LockDoc, welche sich nicht innerhalb von LockDoc lösen lassen. Dazu gehören aufwändigere Synchronisationsmechanismen wie die Referenzzählung (vgl. Abschnitt 5.1), aber auch Unklarheiten darüber, ob gewisse Operationen atomar sind (vgl. Unterabschnitt 7.4.2) oder Unklarheiten in der Dokumentation selbst (vgl. Unterabschnitt 7.4.1 und Unterabschnitt 7.5.1).

3.7 ZUSAMMENFASSUNG

Es wurde aufgezeigt, dass trotz der Stärken von LockDoc noch einige Stellen existieren, in denen es zu Problemen bei dessen Einsatz kommen kann. Im weiteren Verlauf dieser Arbeit wird geprüft, inwiefern diese Einschränkungen auf das hier untersuchte Betriebssystem zutreffen und wie diese die erzielten Ergebnisse beeinflussen bzw. behindern. Anschließend wird versucht, die relevanten Probleme zu beheben um LockDoc für zukünftige Arbeiten zu verbessern.

4

VERWANDTE ARBEITEN

Es existieren verschiedenste Arbeiten, welche sich bereits mit der Sperrenanalyse in Betriebssystemkernen durch unterschiedliche Methoden befasst haben. Es folgt eine Auswahl dieser, in welcher einerseits die Sperren-Analyse durch LockDoc erweitert wird, aber auch auf andere, alternative Methoden geblickt wird.

4.1 ARBEITSLASTGENERIERUNG ÜBER FUZZING

Wie bereits beschrieben hängt der Erfolg der Nutzung von LockDoc davon ab, ob Code, welcher Fehler beinhaltet, auch ausgeführt wird. Wird dieser nicht ausgeführt, so können in der Nachverarbeitung der Daten keine Informationen über dort gesetzte Sperren gewonnen werden. Daher ist es imperativ, dass während des Experiments im zu analysierenden Betriebssystem eine Arbeitslast ausgeführt wird, welche möglichst viele unterschiedliche Kernel-Funktionen aufruft.

In vorangegangenen Arbeiten, welche sich mit LockDoc beschäftigen, wurde oftmals auf das LTP zurückgegriffen, welches eine Testsuite für Linux Kernel Komponenten darstellt (vgl. Unterabschnitt 5.3.2). Im Bezug auf das VFS-Subsystem erreicht LTP jedoch nur eine Basisblock-Abdeckung von etwa 34,7%. [LTS20]

Lochmann *et. al.* untersuchen in [LTS20] eine Alternative zu diesen „klassischen" Methoden, indem sie *Syzkaller* zur Generierung einer Arbeitslast nutzen.

Syzkaller ist ein „unsupervised coverage-guided kernel fuzzer". Ursprünglich ist es als Werkzeug gedacht, um Bugs in diversen Kernen zu finden, indem es diese von der Anwendungsebene aus zu Abstürzen oder anderem unerwarteten Verhalten bringt. Dies geschieht, indem es Programme generiert, welche Kernel-Schnittstellen mit zufälligen Parametern aufrufen. Dabei zielt es darauf ab, eine möglichst hohe Abdeckung an Basisblöcken im Kern zu erreichen.

Lochmann *et. al.* haben Syzkaller so modifiziert, dass es zwar Programme generiert, aber nicht darauf abzielt, mit Hilfe dieser den Kern zum Absturz zu bringen. Stattdessen

```
int main(void)
{
        syscall(__NR_mmap, 0x1ffff000, 0x1000, 0, 0x32, -1, 0);
        syscall(__NR_mmap, 0x20000000, 0x1000000, 7, 0, x32, -1, 0);
        syscall(__NR_mmap, 0x21000000, 0x1000, 0, 0x32, -1, 0);

        *(uint32_t*)0x20002480 = 0x20000340;
        memcpy((void*)0x20000340, "\x12", 1);
        *(uint32_t*)0x20002484 = 1;
        *(uint32_t*)0x20002488 = 0;
        syz_read_part_table(0, 1, 0x20002480);
        return 0;
}
```

Listing 1: Beispiel für eines der durch Syzkaller generierten Programme, aus [LS21].

wird der Fuzzing-Algorithmus genutzt, um Programme zu generieren, welche zuvor nicht abgedeckte Kernel-Basisblöcke aufrufen. Wird so ein Programm gefunden, wird es für die spätere Ausführung innerhalb der LockDoc Experimente aufgezeichnet. Ein Beispiel für solch ein Programm ist in Listing 1 zu sehen. Durch diesen Ansatz kann eine höhere Abdeckung erreicht werden, was folglich zu einem optimaleren Datensatz für die eigentliche LockDoc-Analyse führt. [LTS20] Alle hier genanten Experimente sind auf das VFS-Subsystem des Linux-Kernels beschränkt.

Durch diese verbesserte Methode zur Generierung von Arbeitslasten wurde eine Basisblock-Abdeckung des VFS von 43,8% in Kombination mit LTP und damit eine Basisblock-Abdeckung des gesamten Kernels von 10.0% erreicht. [LS21]

4.2 PORTIEREN VON LOCKDOC AUF WEITERE BETRIEBSSYSTEME

Nach dem Erfolg von LockDoc zum Auffinden von möglichen Bugs im Linux Kernel, wurde in [LS21] evaluiert, ob und wie sich LockDoc auf weitere Betriebssysteme, insb. FreeBSD, portieren lässt.

Als Begründung zur Auswahl von FreeBSD nennen die Autoren primär die „saubere“ Implementierung des Betriebssystems und die detailliertere Dokumentation im Bezug auf korrekte Nutzung von Sperren.

Im Rahmen der Portierung von LockDoc auf FreeBSD wurden die Analysewerkzeuge dahingehend erweitert, als dass diese sich existierende Dokumentation zunutze machen können, insb. zur besseren Auswahl von Strategien zur Hypothesengenerierung.

Die Portierung selbst wurde in das in FreeBSD existierende *Witness*-System, welches eine Kernel-Komponente zur besseren Analyse von Sperrennutzung ist, integriert. Dadurch konnte der Aufwand der Instrumentierung von FreeBSD vermindert werden.

Auch auf FreeBSD konnten durch LockDoc einige Fehler in der Implementierung des FreeBSD-Kerns gefunden werden, aber auch Sperren-Regeln, welche sich (noch) nicht durch LockDoc analysieren lassen. Dazu gehört der bereits angesprochene Fall der freien Auswahl zwischen mehreren Sperren zum Zugriff auf eine Datenstruktur.

4.3 ALTERNATIVEN ZUR POST-MORTEM ANALYSE

Im Jahr 2003 stellten Engler *et. al.* in [EA03] eine Methode zum automatisierten Auffinden von Race Conditions und Deadlocks vor. RacerX arbeitet nach dem bereits vorgestellten Prinzip der statischen Analyse.

Das Hauptaugenmerk liegt auf der Eliminierung einiger der genannten Nachteile der statischen Analyse, insb. werden die folgenden Ziele gesetzt:

1. Minimierung der nötigen Annotationen (und des damit verbundenen manuellen Arbeitsaufwandes). Der Aufwand der Annotation beschränkt sich hierbei auf unter 100 Zeilen an zusätzlichem Code für jeweils Linux und FreeBSD. Ähnlich zu LockDoc müssen primär Routinen, welche Sperren holen oder freigeben, annotiert (bzw. instrumentiert) werden.

2. Einstufung der Wichtigkeit der gefundenen Fehlerkandidaten. Dabei werden zum Abschluss der Analyse für alle gefundenen Kandidaten Faktoren in Betracht gezogen, wie die Höherpriorisierung von globalen Sperren, die Länge des Call Stacks oder der Anzahl der Threads. Dies soll zu einer höheren Effektivität beim Einsatz von RacerX führen.

3. Minimierung der false Positives. Diese treten primär durch inkorrekte Beschränkungen bzgl. der Sperren auf, somit wurde in der Arbeit ein Hauptaugenmerk darauf gelegt, dass diese nicht auftreten. Erreicht wird dies durch die gesonderte Analyse von Semaphoren, welche nicht nur zum gegenseitigen Ausschluss ein-

gesetzt werden, die Berücksichtigung des BKL (vgl. Abschnitt 2.1) und weiteren Techniken.

4. Skalierbarkeit. Auf die genauere Umsetzung dieser Eigenschaft wird in der Arbeit allerdings nicht weiter eingegangen.

Auf einer abstrakten Ebene basiert die Vorgehensweise von RacerX auf den folgenden Schritten:

1. Manuelles Spezifizieren von Sperrenoperationen und Interrupt-manipulierenden Funktionen, sowie optionales Markieren von Single- bzw. Multi-Threaded Routinen und Routinen zur Unterbrechungsbehandlung.
2. Generieren eines Kontrollflussdiagramms (engl. Control Flow Graph, CFG).
3. Traversieren des CFG unter Simulation aller möglichen Sperren und weiterleiten aller Statements an Algorithmen zur Untersuchung auf Verklemmungen und Race Conditions.
4. Sortieren der gefundenen Fehler anhand ihrer Schwere.
5. Manuelles Inspizieren der möglicherweise problematischen Codepfade.

Zur Erkennung von Verklemmungen und Data Races wird in RacerX auf den bereits in Unterabschnitt 2.4.4 vorgestellten LockSet-Algorithmus zurückgegriffen.

Während RacerX es zwar erreicht, einen Teil der Probleme, welche mit statischer Analyse verbunden sind, zu umgehen, so leidet es dennoch unter einigen der üblichen Einschränkungen wie der fehlenden Alias-Analyse.

4.4 ZUSAMMENFASSUNG

Es existieren unterschiedliche Möglichkeiten, Programmcode auf Fehler bzgl. seiner Sperren-Operationen zu analysieren, allerdings ist keine dieser Methoden ohne ihre Schwachstellen. Ziel dieser Arbeit ist es daher, den existierenden LockDoc-Ansatz zur Sperren-Analyse in Betriebssystemkernen dahingehend zu erweitern, als dass dieser mögliche Fehler in weiteren Betriebssystemkernen findet. Dabei wird besonderer Wert darauf gelegt, die Stärken des genutzten Betriebssystems dazu zu nutzen, um LockDoc selbst für zukünftige Arbeiten zu verbessern.

5

NETBSD

In dieser Arbeit wird mit Hilfe des LockDoc-Ansatz ein Betriebssystems evaluiert, unter der Hoffnung, die beiden Projekte so verbessern zu können. Lochmann *et. al.* haben LockDoc in [LSBS19] bereits für Linux und in [LS21] bereits für FreeBSD genutzt. Ein wichtiges Ziel der Arbeit besteht allerdings auch daraus, den LockDoc-Ansatz selbst zu verbessern.

Insbesondere durch die Tatsache, dass für manche Betriebssysteme stringente Kernel-Dokumentation im Bezug auf etwaiges Locking existiert und eine hohe Wertschätzung der Korrektheit des Codes liegt, besteht die Hoffnung, dass die eigentliche Implementierung dieser folgt und durch diesen Zusammenschluss aus konsistentem Locking und dessen Dokumentation das Hauptaugenmerk auf die Verbesserung des LockDoc-Ansatzes gelegt werden kann.

Für diese Arbeit wird also ein Betriebssystem mit einem Kern benötigt, welcher noch nicht in Verbindung mit LockDoc genutzt wurde, durch LockDoc verbessert werden kann und gleichzeitig diese genannte, stringente Synchronisation und Dokumentation verspricht. Eine Kombination dieser Faktoren sollte dazu in der Lage sein, den Ansatz selbst zu evaluieren und ggf. zu optimieren.

Als solche Kombination aus Betriebssystem und Kernel wird NetBSD gewählt.

Die Auswahl von NetBSD als Zielsystem hat verschiedenste Gründe, im Folgenden werden die Ausschlaggebenden benannt.

Es wird mit den wichtigsten Argumenten begonnen:

Aufbau

Im Vergleich zu anderen Betriebssystemen (insb. Linux) ist NetBSD relativ einfach aufgebaut. Während der Linux-Kernel in der aktuellen Version etwa 36,5 Millionen

Zeilen Code beinhaltet, so besteht der NetBSD Kernel[1] aus lediglich 11,5 Millionen Zeilen Code. Dadurch ergibt sich auch eine geringere Anzahl an Personen, welche verantwortlich für die Entwicklung des Codes sind, was zur Folge hat, dass dieser (in der Theorie) besser zusammenhängend ist. Auch viele Kommandozeilenwerkzeuge gehören direkt zu NetBSD und werden nicht separat vom Kernel selbst entwickelt. Die Codebasis ist kompakt genug, als dass eine einzelne Person einen Großteil dieser überblicken kann.

Dokumentation

Ein weiterer Vorteil von NetBSD ist die Existenz eines neunten Handbuch-Abschnittes. Während Linux Handbuchseiten von Abschnitt 1 (*User Commands*) bis Abschnitt 8 (*System Administration Tools and Daemons*) reichen, so existiert in NetBSD zusätzlich noch Abschnitt 9 (*Kernel Internals*). Dieser beinhaltet viel Dokumentation zur eigentlichen Funktionsweise des Kernels, oftmals auch mit relevanten Begründungen, Erläuterungen und Empfehlungen. Des Weiteren befindet sich nach Abschnitt 9 noch „9lua“ (*LUA Kernel Bindings*), welcher in dieser Arbeit allerdings keine Relevanz findet. Linux (und FreeBSD) besitzen zwar ähnlich geartete Dokumentation, diese ist allerdings nicht in das System integriert. Sie befindet sich in Wikis, Büchern und weiteren Formaten. Das macht sie schwerer aufzufinden. Diese Externalität sorgt auch für eine größere Differenz zwischen dem eigentlichen Kernel und dieser erweiterten Dokumentation.

Projektbeschreibung und -ziele

Weiterhin wichtig sind die Projektbeschreibung und -ziele, mit denen die NetBSD Foundation das Projekt identifiziert. So wird auf der entsprechenden Internetseite[2] aufgezählt, dass das Projekt Wert auf „korrektes Design und gut geschriebenen Code legt“. Ein sehr relevantes Zitat aus der selben Quelle lautet:

> Some systems seem to have the philosophy of “*If it works, it's right*”. In that light NetBSD could be described as “*It doesn't work* ***unless*** *it's right*”.

[1] Gemessen anhand des `sys`-Ordners des NetBSD Source Trees
[2] https://www.netbsd.org/about/system.html

Insbesondere dadurch, dass das NetBSD Projekt der Richtigkeit des Codes eine hohe Priorität zuweist, eignet es sich als ideale Testumgebung zur Evaluation von LockDoc.

Neben diesen ausschlaggebenden Gründen existieren noch weitere, untergeordnete Argumente für NetBSD, welche diese Entscheidung weiter bestätigen:

Alter/Verbreitung

NetBSD besitzt eine vergleichsweise alte Codebasis. Es wurde 1993 auf Basis von 386BSD gegründet (welches wiederum auf dem zuerst 1978 veröffentlichtem BSD basiert) und erhält seitdem konstante Verbesserungen. Neben Linux und FreeBSD gehört es zu den am meisten verbreiteten Open-Source Betriebssystemen und wird dadurch seit Jahrzehnten in der Praxis getestet. Zum Zeitpunkt der Veröffentlichung dieser Arbeit ist NetBSD 9.3 die aktuelle stabile Version, NetBSD 10 befindet sich in der Beta-Phase.

Open Source

Der NetBSD-Kernel ist (bis auf wenige Ausnahmen) unter einer Berkeley-Lizenz[3] veröffentlicht, welche es einerseits ermöglicht, den kompletten Quelltext einzusehen und weiterhin erlaubt, eben diesen Quelltext nach Belieben (unter Beibehaltung der Lizenz) zu modifizieren. Letzteres ist wichtig, um die Experimente zur Analyse des Lockings ausführen lassen zu können.

Andauernde Unterstützung/Entwicklung

NetBSD wird – trotz seines relativ hohen Alters – immer noch unterstützt und weiterentwickelt. Somit können etwaige Bugs, welche im Laufe dieser Arbeit gefunden werden, an die Mitwirkenden gemeldet und ggf. sogar behoben werden.

Unerforscht

Die in dieser Arbeit beschriebenen LockDoc Experimente basieren auf vorangegangenen Forschungen. [LSBS19, LS21] In diesen wurden bereits Linux 5.4 und FreeBSD 13

[3] https://www.netbsd.org/about/redistribution.html

hinsichtlich ihrer Synchronisation analysiert. Durch die Nutzung von NetBSD wird keine redundante Arbeit vollbracht. Stattdessen werden neue Forschungsziele erreicht, welche es ermöglichen, NetBSD mit Linux und FreeBSD zu vergleichen. Daraus lassen sich auch Rückschlüsse bezüglich der verschiedenen Entwicklungsmodelle und Codebasis-Größen ziehen.

5.1 ARTEN VON SPERREN

Der NetBSD Kernel nennt in Version 10.99.5 insgesamt 12 verschiedenen Gruppen von Methoden, welche zur Zugriffskontrolle im Betriebssystemkern genutzt werden können. Diese Liste ist in den Handbuchseiten über `man 9 locking` einsehbar.

Im Folgenden befindet sich diese Auflistung mit weiteren Erläuterungen und Ausführungen darüber, wozu diese Locking-Methoden genutzt werden können und inwiefern sie relevant für diese Arbeit sind.

An dieser Stelle ist wichtig zu erwähnen, dass LockDoc implementiert wurde, um auf *gegenseitigen Ausschluss* (engl. mutual exclusion) hin zu überprüfen. Einige der folgenden Methoden hingegen stellen *einseitige Synchronisierung* dar, d. h. der jeweilige Synchronisationsmechanismus wird nur an einer von mehreren Stellen genutzt, welche auf eine Datenstruktur zugreifen. Die folgenden Beschreibungen entstammen wenn nicht anders angegeben aus den jeweiligen NetBSD Handbuchseiten.

`ATOMIC_OPS(9)` Der NetBSD Kern implementiert atomare Operationen in jeweiligem architekturspezifischem Assembly-Code. Für die zu analysierende i386 bzw. x86-Architektur wird zur Implementierung der atomaren Operationen die `lock`-

```
ENTRY(_atomic_add_32)
        movl        4(%esp), %edx
        movl        8(%esp), %eax
        LOCK
        addl        %eax, (%edx)
        ret
END(_atomic_add_32)
```

Listing 2: Implementierung von `_atomic_add_32`, aus `common/lib/libc/arch/i386/atomic/atomic.S`

Instruktion benutzt (vgl. Listing 2). Diese stellt sicher, dass während der Operation exklusiver Zugriff auf die relevanten Speicherbereiche besteht. Zusätzlich wird davon ausgegangen, dass Speicherzugriffe in Wortgröße allgemein atomar sind. Da dies eine Form der einseitigen Synchronisierung darstellt, ist diese Form von Operationen nicht relevant für diese Arbeit.

`CONDVAR(9)` Während ein Mutex anhand des Zugriffes auf gewisse Datenstrukturen sperrt, sperren Conditional Variables (Bedingungsvariablen) anhand des eigentlichen Inhaltes dieser Datenstrukturen. In NetBSD werden sie meist zur Synchronisation von Zugriffen auf begrenzte Ressourcen genutzt, bspw. Systemressourcen wie Speicher. [Dev20] Da diese Bedingungsvariable aber eingesetzt werden, indem sie nur in Verbindung mit gewöhnlichen Mutexen agieren, ist es nicht nötig, sie explizit zu Analysieren – Zugriff auf durch Conditional Variables gesperrte Datenstrukturen geschieht implizit bei der Analyse der allgemeinen Mutexe. Außerdem stellen sie keine Form des gegenseitigen Ausschlusses dar, sind also irrelevant im Bezug auf LockDoc.

`MEMBAR_OPS(9)` Es gibt verschiedenen Ursachen, aus denen eine lineare Abfolge von Programmcode nicht-linear ausgeführt werden kann, dazu gehören oftmals CPU-interne Optimierungen zur Laufzeit. [BC05] Intern rufen die zugehörigen Funktionen zum Verwalten der Speicherbarrieren direkt architekturabhängige Assembly-Routinen auf. Da diese nicht zum gegenseitigen Ausschluss dienen, ist diese Art von Locking in dieser Arbeit nicht relevant.

`MUTEX(9)` Ein Mutex, innerhalb des NetBSD Kernels als `kmutex_t` bezeichnet, stellt den primitivsten Weg dar, wechselseitigen Ausschluss (engl. Mutual Exclusion) zu garantieren. Diese Datenstruktur ist relevant für die LockDoc Analyse.

`RWLOCK(9)` Das RWLock, innerhalb des NetBSD Kernels als `krwlock_t` bezeichnet, ist neben dem bereits genannten Mutex die zweite Locking-Primitive. Im Gegensatz zum Mutex kann man durch das RWLock allerdings zwischen Leser- und Schreiber-Sperren differenzieren. Da diese Art von Sperren zum gegenseitigen Ausschluss dient ist diese Datenstruktur relevant für die LockDoc Analyse.

`SOFTINT(9)` Diese Klasse von Funkionen dient zum etablieren von Software Interrupts. Dieses Etablieren von Unterbrechungen zur späteren Ausführung ist selbst nicht

relevant für die in dieser Arbeit betriebene Analyse, anders als das explizite Verhindern derer Ausführung, bspw. über die `spl`-Methoden.

`SPL(9), SPLRAISEIPL(9)` Funktionen, welche die Prioritäten von Unterbrechungen verändern und diese damit ggf. verhindern, sind indirekt relevant für die LockDoc-Analyse: Da durch die Erhörung des Interrupt-Levels sichergestellt werden kann, dass der derzeit ausgeführt Code nicht durch eintreffende Interrupts unterbrochen werden kann, kann dies als eine Form des Sperrens der aktiven Datenstrukturen vor Veränderungen durch andere Prozesse angesehen werden.

`PSERIALIZE(9)` Ein passiver Mechanismus zur Serialisierung von Lese-Operationen. [Dev16a] Sie benötigen keine Interprozessorsynchronisation, außerdem werden während ihrer Ausführung Soft-Interrupts deaktiviert. Sie sind nicht für langandauernde Aufgaben geeignet. Da es sich um eine Methode zur Serialisierung handelt, ist diese nicht relevant für LockDoc.

`PSREF(9)` Beschrieben als „Mittelpunkt“ zwischen `pserialize` und gewöhnlicher Referenzzählung ermöglicht diese Klasse von Methoden das effiziente Verwalten von Referenz-Countern. [Dev16b] Obwohl Referenzzähler relevant für die Evaluation via LockDoc sind, so unterstützen die entsprechenden Werkzeuge bisher deren Nutzung noch nicht (vgl. Abschnitt 3.6). Dementsprechend wird ihnen in dieser Arbeit keine Beachtung geschenkt.

`LOCALCOUNT(9)` Localcounts stellen eine weitere Methode zur Referenzzählung dar, sie unterscheiden sich von den atomaren Operationen und Alternativen wie `psref` lediglich primär durch die Effizienz der Operationen. Es trifft die selbe Argumentation wie für `psref` zu, daher wird auch dieser Synchronisationsmechanismus hier nicht näher betrachtet.

`WORKQUEUE(9)` Diese Menge an Routinen dient der Verwaltung von Arbeitslasten innerhalb eines Threads. Auch hier handelt es sich nicht um eine Methode zur gegenseitigen Synchronisation.

5.2 SUBSYSTEME

Die Auswahl von entsprechenden Subsystemen des Kernels ist elementar für den Erfolg des vorgestellten Ansatzes. Bei dessen Auswahl ist allerdings zu beachten, dass nicht

jedes Subsystem sich dazu eignet, durch LockDoc (oder durch Post-Mortem-Analyse im Allgemeinen) genauer analysiert zu werden.

5.2.1 *Auswahlkriterien*

Wichtig bei der Auswahl eines solchen Subsystems ist das Beachten einiger wichtiger Einschränkungen, die bezüglich der LockDoc-Tests gelten:

KEINE ECHTZEITANFORDERUNGEN Die Arbeitslasten werden mit Hilfe des FAIL*-Emulators auf einem emulierten Kernel in einem emulierten x86-System ausgeführt. FAIL* basiert auf Bochs, welches für x86 notwendige Geräte wie Timer möglichst akkurat, allerdings nicht in Echtzeit emuliert. Dadurch kann es abhängig von anderen Prozessen auf dem Host zu Schwankungen in der Ausführungsgeschwindigkeit von FAIL* und damit Inkonsistenzen des internen Timers und anderen Geräten kommen. Es können somit keine Garantien bzgl. uniformer Geschwindigkeit aufgestellt werden. Dies verhindert das Testen von Teilen des Kernels, welche eine Synchronisation des internen Timers mit anderen, externen Systemen benötigen. Subsysteme wie das Netzwerk oder das Management weiterer Hardware, welche unabhängige Timer beinhaltet, entfällt somit, sofern dieses nicht vollständig mit-emuliert werden kann.

BESCHRÄNKTE HARDWARE Der Bochs-Emulator zielt nicht darauf ab, eine große Anzahl an unterschiedlicher Hardware zu emulieren. Stattdessen ist sein Ziel, möglichst akkurate Emulation zu bieten. Aus diesem Grund entfällt das Testen von Subsystemen, welche Treiber für Hardware darstellen, da diese meist nicht in Bochs implementiert ist.

ARBEITSLAST Um ein Subsystem adäquat evaluieren zu können, bedarf es einer Arbeitslast, welche möglichst viele Codepfade des Subsystems ausführen kann. Dies vermindert die Anzahl der zur Verfügung stehenden Subsysteme erneut, allerdings ist diese Verminderung dank der Existenz des Automated Testing Framework (ATF) (vgl. Unterabschnitt 5.3.3) geringer als in anderen Betriebssystemen.

KOMPAKTHEIT Das zu testende Subsystem sollte innerhalb des Quellcodes möglichst strikt von anderen Teilen des Kernels differenziert sein. Dies vereinfacht die Instrumentierung und vermindert damit das Risiko an Fehlern in dieser.

An dieser Stelle ist anzumerken, dass das Vorhandensein von etwaiger Dokumentation bzgl. des Lockings kein striktes Ausschlusskriterium ist, ihr Vorhandensein hat lediglich Einfluss auf die zur Verfügung stehenden Analysemethoden (vgl. Abschnitt 3.1). Existiert diese, so kann sie durch LockDoc auf Richtigkeit überprüft werden, außerdem kann durch diese der Algorithmus zur Hypothesengenerierung evaluiert werden (vgl. Abschnitt 7.3). Existiert diese Dokumentation jedoch nicht, so kann sie mit Hilfe von LockDoc ergänzt werden.

5.2.2 *VFS-Subsystem*

Unter Berücksichtigung der vorherigen Kriterien und Abgleich dieser mit den in NetBSD verfügbaren Subsystemen ergibt sich das VFS-Subsystem als geeigneter Kandidat zur genaueren Analyse.

> The kernel's Virtual File System (VFS) subsystem provides access to all available file systems in an abstract fashion, just as vnodes do with active files. Each file system is described by a list of well-defined operations that can be applied to it together with a data structure that keeps its status. [Dev11, Kap. 2]

Der Zweck des VFS-Subsystems ist es, eine Dateisystem-unabhängige Schnittstelle zur Verwaltung von Dateien zur Verfügung zu stellen. In früheren Versionen von BSD-Systemen existierte dieses Subsystem nicht, der Zugriff auf Dateien geschah direkt anhand der auf Festplatten gespeicherten Inodes. Während dieser Ansatz damals funktioniert hatte, so ist er heutzutage aufgrund der immer größeren Anzahl an unterschiedlichen Dateisystemen nicht mehr praktikabel. Aus diesem Grund hat Sun Microsystems eine zusätzliche Abstraktionsschicht, die Vnodes implementiert. [Chu16] Diese oder vergleichbare Implementierungen finden sich bis heute in diversen Unix-ähnlichen Betriebssystemen. [BC05]

Das VFS-Subsystem bietet die idealen Voraussetzungen, um mit Hilfe von LockDoc überprüft zu werden. Es befindet sich vollständig innerhalb der bereits gestellten Anforderungen. Zum einen ist es dadurch, dass es lediglich als Art Bindeglied zwischen dem Kernel und den jeweiligen Dateisystem-Implementierungen fungiert, unabhängig von jeglichen Echtzeitanforderungen. Anfragen an die zur Verfügung gestellten Festplatten werden innerhalb von Bochs emuliert und müssen somit nicht mit externen Systemen synchronisiert werden. Dadurch, dass dieser Prozess vollständig innerhalb von FAIL*

bzw. Bochs stattfindet, wird auch die Anforderung auf den Verzicht von besonderer Hardware erfüllt. Wie bereits erwähnt existieren auch adäquate Arbeitslasten, auf diese wird genauer in Abschnitt 5.3 eingegangen. Die Kompaktheit des VFS-Subsystems ist dadurch gegeben, dass es vollständig innerhalb der `sys/kern/vfs_*.c` Dateien (und zugehörigen Headern) implementiert ist.

Des Weiteren kann dadurch, dass es sich um eine Schnittstelle zu den eigentlichen Dateisystem-Implementierungen handelt durch einen geringen Instrumentierungsaufwand dieses Interfaces die Menge aller (testbaren) Dateisysteme überprüft werden.

Außerdem eignet es sich gut zur Analyse in dieser Arbeit, da es eine gute Vergleichbarkeit zu bereits existierenden Testergebnissen aus Experimenten, welche sich mit Linux und FreeBSD beschäftigt haben, bietet.

5.2.2.1 Datenstrukturen

Das VFS-Subsystem in NetBSD besitzt primär vier stark zusammenhängende Datenstrukturen. Viele von ihnen enthalten zusätzlich klar definierte Regeln bzgl. des Sperrens der einzelnen Mitglieder:

`struct vnode_impl` Diese Datenstruktur stellt eine Art übergeordnete Datenstruktur über die eigentliche `vnode` dar. Neben dem Mitglied `vi_vnode`, welches die `vnode` selbst beinhaltet, enthält sie u. a. Informationen zur Verwaltung der `vnode`s, Caches und Indikatoren ob die `vnode` *clean* oder *dirty* ist, d. h. ob sie verändert wurde und auf das Speichermedium geschrieben werden muss. In der Regel wird ein Objekt vom Typ `vnode`, welches in den Speicher geladen wird, immer in eine `vnode_impl` Struktur eingebettet.[4] Die Datenstruktur besitzt drei eingebettete Sperren, für jedes Member existiert ein Kommentar darüber, welche Locking-Vorkehrungen vor dem Zugriff getroffen werden müssen (vgl. Listing 3).

`struct vnode` Das `vnode` besteht aus den Daten, auf welche von Applikationen der Anwendungsebene zugegriffen werden kann, sowie aus einigen Mitgliedern zwecks Garantie der korrekten Synchronisation. Der eigentliche Inhalt des `vnode` befindet sich in dem `v_uobj`, einer Struktur des Types `uvm_object`. Ähnlich wie

[4] Innerhalb des Kernels werden oftmals die C-Makros `VIMPL_TO_VNODE` und `VNODE_TO_VIMPL` benutzt, um von einer Datenstruktur auf die jeweils andere zugreifen zu können.

in `vnode_impl`, ist für jedes Mitglied individuell dokumentiert, welche Sperren vor dem Zugriff geholt werden müssen. Es ist anzumerken, dass das `vnode` auch spezielle Arten von Dateien wie Mounts (s. u.), Sockets, Geräte oder FIFOs über eine C Union darstellt (vgl. Unterunterabschnitt 6.2.1.1).

`struct mount` Beschreibt einen Mount. Dieser besitzt drei Zeiger zu Sperren (vgl. Unterabschnitt 6.3.2), im Vergleich zu den anderen Datenstrukturen allerdings keine Dokumentation über die notwendige Synchronisation.

`struct buf` Eine allgemeinere Datenstruktur, welche für eine Vielzahl an I/O-Operationen im Kernel genutzt wird. Wie `vnode` besitzt sie eine C Vereinigung, neben Zeigern zu Mutexen allerdings relativ primitive Elemente. Sie eignet sich dahingehend zur genaueren Analyse, als dass Zugriffe auf Buffer oftmals parallel stattfinden, aber auch die enge Integration mit den anderen Datenstrukturen erscheint interessant. So liest ein Kommentar der Buffer-Datenstruktur:

> For buffers associated with a vnode, b_objlock points to vp->v_interlock.
> If not associated with a vnode, it points to the generic buffer_lock.[5]

Für alle Datenstrukturen existieren weitgehend äquivalente Implementierungen in Linux und FreeBSD.

5.2.2.2 *Sperren*

Wie für andere Subsysteme, sind innerhalb des VFS drei Arten von Sperren relevant (vgl. Abschnitt 3.2):

1. Direkt in eine Datenstruktur eingebettete Sperren, welche oftmals die gesamte Datenstruktur oder eine Menge an ihren Mitgliedern vor Zugriff schützt. Ein Beispiel hierfür ist `vnode_impl.vi_lock`.

2. Statische Sperren, welche zum Start/zur Initialisierung des Kernels alloziert werden. Von diesen Existiert jeweils genau eine Instanz pro ausgeführtem Kernel. Das `buffer_lock` wird bspw. genutzt, wenn ein Buffer keiner spezifischeren Sperre zugewiesen werden kann.

[5] https://nxr.netbsd.org/xref/src/sys/sys/buf.h?r=1.134#105

```
/*
 * Reading or writing any of these items requires holding the appropriate
 * lock.  Field markings and the corresponding locks:
 *
 *        -          stable throughout the life of the vnode
 *        c          vcache_lock
 *        d          vdrain_lock
 *        i          v_interlock
 *        l          vi_nc_listlock
 *        m          mnt_vnodelock
 *        n          vi_nc_lock
 *        n,l        vi_nc_lock + vi_nc_listlock to modify
 *        s          syncer_data_lock
 */
struct vnode_impl {
        struct vnode vi_vnode;
        struct vcache_key vi_key; // c vnode cache key
        struct vnode_klist vi_klist; // i kevent / knote state
        struct vnodelst *vi_lrulisthd; // d current lru list head
        int vi_synclist_slot; // s synclist slot index

        // [...]

        uid_t vi_nc_uid; // n,l cached UID or VNOVAL
        gid_t vi_nc_gid; // n,l cached GID or VNOVAL
        uint32_t vi_nc_spare; // - spare (padding)

        /// [...]

        krwlock_t vi_lock; // - lock for this vnode
        krwlock_t vi_nc_lock; // - lock on node
        krwlock_t vi_nc_listlock; // - lock on nn_list
};
typedef struct vnode_impl vnode_impl_t;
```

Listing 3: Ein Auszug aus der Definition der Datenstruktur `struct vnode_impl`, aus `sys/sys/vnode_impl.h`. Für diese Arbeit irrelevante Kommentare und Compilerinstruktionen wurden entfernt.

```
/*
 * Reading or writing any of these items requires holding the appropriate
 * lock.  Field markings and the corresponding locks:
 *
 *        -          stable, reference to the vnode is required
 *        b          bufcache_lock
 *        e          exec_lock
 *        f          vnode_free_list_lock, or vrele_lock for vrele_list
 *        i          v_interlock
 *        i+b        v_interlock + bufcache_lock to modify, either to
 ↪  inspect
 *        i+u        v_interlock + v_uobj.vmobjlock to modify, either to
 ↪  inspect
 *        k          locked by underlying filesystem (maybe kernel_lock)
 *        u          v_uobj.vmobjlock
 *        v          vnode lock
 *  [...]
 */
struct vnode {
        // [...]

        struct uvm_object v_uobj; // u   the VM object
        voff_t v_size; // i+u size of file
        voff_t v_writesize; // i+u new size after write

        // [...]

        kmutex_t *v_interlock; // -   vnode interlock
        struct mount *v_mount; // v   ptr to vfs we are in
        int         (**v_op)(void *); // :   vnode operations vector
        union {
                struct mount *vu_mountedhere; // v   ptr to vfs (VDIR)
                struct socket *vu_socket; // v   unix ipc (VSOCK)
                struct specnode *vu_specnode; // v   device (VCHR, VBLK)
                struct fifoinfo *vu_fifoinfo; // v   fifo (VFIFO)
                struct uvm_ractx *vu_ractx; // u   read-ahead ctx (VREG)
        } v_un;

        // [...]
};
```

Listing 4: Ein Auszug aus der Definition der Datenstruktur `struct vnode`, aus `sys/sys/vnode.h`. Für diese Arbeit irrelevante Kommentare und Compilerinstruktionen wurden entfernt.

3. Sperren, welche zur Laufzeit alloziert werden, aber nicht direkt in beobachtete Datenstrukturen eingebettet sind, stellen einen signifikanten Teil der im VFS-Subsystem genutzten Sperren dar. Dazu gehören sowohl Zeiger in beobachteten Datenstrukturen, als auch globale Sperren, welche beim Systemstart (bspw. abhängig von der Anzahl der CPU-Kerne) initialisiert werden. Die existierenden LockDoc-Werkzeuge besitzen keinen direkten Weg, mit diesem Sperren umzugehen, vermutlich, da diese in Linux und FreeBSD seltener eingesetzt werden als in NetBSD (vgl. Unterabschnitt 6.3.2).

5.3 ARBEITSLAST

Der beschriebene Ansatz liefert hilfreichere Ergebnisse, wenn bei dem zu analysierenden Durchlauf möglichst viele Codepfade abgedeckt werden (vgl. Abschnitt 4.1). Aus diesem Grund ist die Auswahl einer entsprechenden Arbeitslast elementar zur Erzeugung eines guten Datensatzes. Im Folgenden werden mehrere zur Auswahl stehende Optionen betrachtet.

5.3.1 *LockDoc-Test*

LockDoc-Test ist eine von Lochmann *et. al.* vorgestellte Methode zur Prüfung der Korrektheit des LockDoc-Ansatzes selbst und seiner Implementierung (vgl. Abschnitt 7.1). Neben der Untersuchung der spezifisch für LockDoc-Test modifizierten Datenstrukturen eignet es sich aber auch zur allgemeinen Untersuchung einiger Subsysteme. Im Bezug auf das VFS-Subsystem wird beim Start des Betriebssystems auf eine gewissen Anzahl an Dateien zugegriffen, welche für die korrekte Initialisierung des Betriebssystems notwendig sind (bspw. das Init-System, Konfigurationsdateien für diverse Dienste, Shell- und Login-Binaries). Alleine durch diese Zugriffe können erste Ergebnisse gewonnen werden, diese sind jedoch bei weitem nicht repräsentativ. Sie dienen lediglich zur schnelleren Entwicklung.

5.3.2 *Linux Test Project*

Das LTP[6] stellt eine in dieser und in vorangegangenen Arbeiten genutzte Methode zum Testen von Linux-Systemaufrufen auf Korrektheit dar.

Durch seine Nutzung in den vorausgegangenen LocкDoc-Experimenten besteht ein weiterer Vorteil daraus, dass die Ergebnisse gut vergleichbar mit denen von Linux und FreeBSD sind. Ein weiterer Vorteil besteht aus der starken Modularität von LTP, so können ohne großen Arbeitsaufwand einzelne Tests hinzugefügt oder (in der vorgestellten Umgebung) fehlerhafte Tests entfernt werden.

Zu beachten ist allerdings, dass dies ein Linux-Projekt ist. NetBSD bietet (analog zu FreeBSD) eine Linux-Emulationsschicht an, diese bezeichnet sich selbst als Linux Kernel 3.11.6 (unter NetBSD 10.99.5). Dies ist eine relativ alte Kernel-Version,[7] was dazu führt, dass einige der LTP-Tests nicht ausgeführt werden können. Außerdem hat das alleinige Nutzen der Linux-Kompatibilitätsschicht den erheblichen Nachteil, dass die dort zu Verfügung gestellten Operationen erstens durch eine weitere Abstraktionsschicht an den NetBSD Kernel weitergeleitet werden (und somit evtl. andere Codepfade nutzen als gewöhnlicher NetBSD-Code) und zweitens diese Operationen nur auf eine Teilmenge der in NetBSD implementierten Kernel-Funktionen abgebildet werden.

Während LTP also im Allgemeinen ein gutes Werkzeug zur Evaluation ist, so sollten zusätzlich für NetBSD native Arbeitslasten evaluiert werden.

5.3.3 *Automated Testing Framework*

> The Automated Testing Framework (ATF) is a collection of libraries and utilities designed to ease unattended application testing in the hand of developers and end users of a specific piece of software. [Dev10]

Das ATF kann als NetBSD-Äquivalent des für Linux geschriebenen LTP angesehen werden. Dadurch, dass es nativ für NetBSD entwickelt wird, steht es in einigen Hinsichten im direkten Gegensatz zum LTP und eliminiert so einen Großteil der oben genannten Nachteile. Insbesondere dadurch dass es direkt in NetBSD integriert ist und parallel zum eigentlichen Kernel weiterentwickelt wird, deckt es eine breite Masse an

[6] https://linux-test-project.github.io/

[7] Linux Kernel Version 3.11.6 wurde 2013 veröffentlicht.

unterschiedlichen Code-Pfaden ab. Der einzige Nachteil gegenüber dem LTP ist, dass das ATF nicht die Linux-Kompatibilitätsschicht selbst testet, was zu unabgedecktem Code im Kernel führen kann, da dieser für die unterschiedlichen Systemaufrufe voneinander abweicht. Die unterschiedlichen Aufrufe der selben Kernel-Methoden sind jedoch von geringer Bedeutung, da ein Großteil der relevanten Sperrenoperationen innerhalb der ursprünglich aufgerufenen Systemaufruf-Routinen geschieht.

Es ist analog zum LTP sehr modular aufgebaut, es kann also genutzt werden um nur einzelne Subsysteme oder gar einzelne Methoden innerhalb dieser Subsysteme zu evaluieren. Daher ist es idealer Kandidat zum Einsatz in Verbindung mit LockDoc.

5.4 ZUSAMMENFASSUNG

In diesem Kapitel wurden sowohl ein Betriebssystem, als auch eines seiner Subsysteme für den weiteren Verlauf begründet ausgewählt.

Zur Umsetzung und Integration von LockDoc in NetBSD ist eine Reihe an Modifikationen nötig. In Abbildung 5.4.1 findet sich eine Übersicht darüber, welche Komponenten des gesamten Systems genauer betrachtet werden müssen.

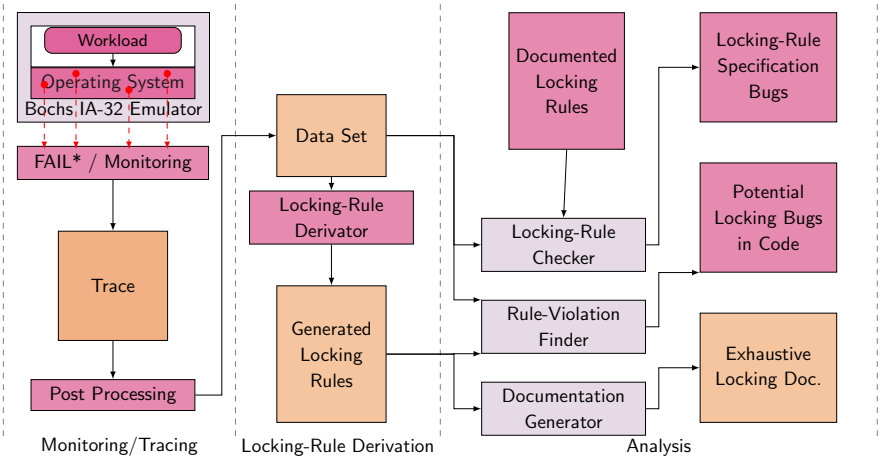

Abbildung 5.4.1: Komponenten der LockDoc-Methode. Magenta hinterlegte Felder müssen im Rahmen dieser Arbeit portiert, angepasst, erweitert oder ausgewertet werden. Als Basis dient Abbildung 2.4.1 aus [Loc21].

6

IMPLEMENTIERUNG

Zur Portierung bzw. Implementierung von LockDoc auf NetBSD ist eine Reihe an Modifikationen an einem Großteil der bisher vorgestellten Werkzeuge und Programme (vgl. Abbildung 5.4.1) nötig. Es folgt eine Übersicht über die allgemein benötigten Modifikationen der LockDoc-Werkzeuge (*FAIL*/Monitoring*, *Post Processing*), sowie ein Einblick in die Modifikation des NetBSD-Kerns und die Instrumentierung der entsprechenden Subsysteme (*Operating System*). Zuletzt wird auf entstandene Herausforderungen eingegangen, gefolgt von einer Analyse des Aufwandes, welcher zur Implementierung von LockDoc auf neuen Betriebssystemen nötig ist.

6.1 MODIFIKATIONEN

Die grundlegende Funktionsweise der LockDoc-Experimente wurde bereits in Abschnitt 2.4 erläutert. An dieser Stelle folgen Erklärungen zu NetBSD-spezifischen Modifikationen.

Basis für die stattfindende Analyse ist NetBSD in Version 10.99.5.[1][2] Es wurde eine Beta-Version von NetBSD 10 ausgewählt, da diese im Vergleich zur letzten stabilen NetBSD Version 9.3 (veröffentlicht August 2022) größere Änderungen in den für diese Arbeit relevanten Subsystemen enthält.[3] Die Codebasen der beiden Versionen trennen sich im Juli 2019, NetBSD 9.3 erhielt von diesem Punkt an nur noch Fehlerverbesserungen, allerdings keine tiefgreifende Veränderungen der einzelnen Subsysteme. Würde in

[1] CVS Aufnahme vom 07.07.2023, Git Commit `88b01cb4e58810347a8cd0b5edf74bd546f8e4c0`

[2] NetBSD bezeichnet Beta-Veröffentlichungen nach dem Schema x.99.y, wobei x das Major Release ist, y bezeichnet die Beta-Iteration.

[3] Dazu gehören das Hinzufügen von Datenstrukturelementen in den zu beobachteten Datenstrukturen, sowie das Aufteilen einiger Mutexe in separate Sperren.

dieser Arbeit NetBSD 9.3 genutzt werden, so wären etwaige hier erzielte Ergebnisse evtl. schon obsolet gewesen.

Damit sich das Betriebssystem für die Analyse durch LockDoc eignet, müssen zunächst einige Modifikationen am Kern getätigt werden.

6.1.1 *Grundlegende Modifikationen*

Eines der ersten Ziele ist es, eine möglichst minimale Kernel-Konfiguration zu erstellen. Dies hat zwei Gründe, einerseits werden so weniger Kernel-Module integriert, welche die (durch die Emulation ohnehin verlangsamte) benötigte Zeit zum Starten des Betriebssystems weiter ansteigen lassen würden. Andererseits werden durch die so fehlenden Module und deren Initialisierungsprozesse die Messergebnisse der eigentlich zu beobachtenden Subsysteme nicht beeinflusst. Außerdem sorgt das Nichtvorhandensein dieser Komponenten dafür, dass weniger Interferenzen aufgrund von zufällig zeitgleich gesetzten Sperren für bestimmte Datenstrukturen existieren (vgl. Abschnitt 7.3). Insgesamt wird die minimalste Kernel-Konfiguration genutzt, welche es ermöglicht das System im Bochs-Emulator (die Basis des FAIL*-Emulators) zu starten, die aber noch die zu untersuchenden Komponenten enthält. Dieser Prozess benötigt auch eine Veränderung der FAIL*/Bochs Konfiguration, welche im Original für Linux/FreeBSD genutzt wurden, da es ansonsten unter NetBSD zu Problemen mit den Grafiktreibern kommt. Diese sind notwendig für etwaiges Debugging der Initialisierungsskripts und der darauffolgenden Arbeitslasten.

Des Weiteren werden Änderungen vorgenommen, um die Benchmarks automatisch von FAIL* starten lassen zu können. Dazu gehören kleinere Modifikationen am Init-System (welches identifiziert, welche Konfiguration der aktuelle Kernel nutzt und im Falle der LOCKDOC-Konfiguration die entsprechenden Arbeitslasten startet) und Veränderungen am Bootloader, sodass die Ausgabe jeglicher Programme über die serielle Schnittstelle statt des emulierten Displays erfolgt. An dieser Stelle ergibt sich eine zusätzliche Hürde dadurch, dass Bochs keine Emulation des Carrier Detect Signals der seriellen Schnittstellen bereitstellt.

6.1.2 *Speicherlayout*

Um die Funktion des FAIL*-Experiments garantieren zu können, benötigt dieses einige Information über das Speicherlayout des laufenden Systems. Die grundlegende

Information über die Position des `extern struct log_action la_buffer`, welches zur Übertragung jeglicher Ereignisse vom NetBSD-Kernel an den Host genutzt wird, wird zur Laufzeit des Experiments aus der Debug-Binary des Kernels extrahiert. Über diese Schnittstelle werden nun weitere benötigte Daten übertragen, dazu gehört:

- Speicherposition des aktuellen Light-Weight Process (LWP).[4]
- Speicherposition der LWP Flags relativ zum LWP
- Speicherposition der Process ID (PID) relativ zum LWP
- Version des genutzten Kernels

Die LWP-bezogenen Informationen werden zur Extraktion von PIDs und zur Identifizierung von Interrupt-Handlern genutzt. Die Version des Kernels wird lediglich zur Identifikation unterschiedlicher Testläufe eingesetzt.

6.1.3 *Modifikationen des Sperrenmanagements*

Damit der LockDoc-Ansatz funktioniert müssen zwei fundamentale Operationen aufgezeichnet werden, damit diese später analysiert werden können: Sperrenoperationen und Speicheroperationen.

6.1.3.1 *Lockdebug Framework*

Ähnlich zu Lockstat in Linux oder dem Witness-System in FreeBSD stellt NetBSD zum Debugging des Sperrensystems das Lockdebug Framework zur Verfügung. Dieses ist eine Kernel Konfigurationsoption (d. h. es wird zum Zeitpunkt der Übersetzung im Kernel aktiviert) und bietet zusätzliche Mechanismen zur Überprüfung von Sperrenoperationen. Dazu gehören Überprüfungen, ob ein Kernel Thread gegen sich selbst sperrt und weitere Überprüfungen auf mögliche Verklemmungen. [Dev17] Außerdem werden durch das Aktivieren dieser Option einige zusätzliche Funktionen und Datenstrukturen zur Verfügung gestellt, über welche Kernel-Entwickler zusätzliche Annahmen (engl.

[4] LWPs sind die in NetBSD genutzte Abstraktionsschicht zur Verwaltung von Prozessen und Threads.

Assertions) aufstellen können. Zusätzlich werden durch dessen Nutzung einige Sperrenoperationen, welche aus Performanzgründen für gewöhnlich in Assembly implementiert sind durch äquivalenten C-Code (für vereinfachtes Debugging) ersetzt.

Anders als in FreeBSD und dessen Witness-System [LS21], ist es in NetBSD nicht direkt möglich, die zu Verfügung gestellten zusätzlichen Hooks der Sperrenfunktionen zur Weiterverarbeitung zu nutzen. Dies rührt daher, dass sämtliche zusätzlichen Funktionen *in* den eigentlichen Sperrenoperationen geschehen, diese aber keine Informationen darüber enthalten, von wo sie aufgerufen wurden (diese Informationen werden in FreeBSD weitergegeben). Lediglich das Ersetzen von Assembly-Routinen durch C-Routinen wird in den LockDoc-spezifischen Kernel-Modifikationen übernommen.

6.1.3.2 *Ablauf der Sperrenaufzeichnung*

Die Lösung für dieses Problem besteht aus dem Einbinden der LockDoc Header Dateien in die Header-Dateien der relevanten Sperren (`sys/sys/mutex.h` und `sys/sys/rwlock.h`). Im LockDoc Header werden die zu beobachtenden Sperren neu definiert (`#DEFINE mutex_enter _mutex_enter`), in den Sperren-Dateien werden die jeweiligen Funktionen so umbenannt, dass keine Neudefinierung stattfindet. Ruft nun ein Kernel Subsystem eine der Sperrenoperationen auf, bspw. über `mutex_enter(lock)`, so wird dies durch die in `lockdoc.h` spezifizierten Makros durch `_mutex_enter(lock, __FILE__, __LINE__, __func__)` ersetzt. Die neu implementierte Funktion `_mutex_enter` schreibt nun die relevanten Daten wie die Art der Sperre, Datei, Zeile und Funktion des Aufrufes in den dafür vorgesehenen Buffer und sendet eine Nachricht an den FAIL*-Client. Diese innere Funktionsweise ist nicht relevant für die eigentliche Analyse, sie liefert allerdings einen Überblick über den Ausmaß des Arbeitsaufwandes welcher entsteht wenn LockDoc auf ein neues Betriebssystem portiert wird.

6.1.3.3 *Unterbrechungsbehandlung*

Ein ähnlicher Prozess findet statt, sobald im Gastsystem Unterbrechungen deaktiviert bzw. wieder aktiviert werden. Da diese von LockDoc als eine Art Sperre interpretiert werden (da sie Nebenläufigkeit temporär verhindern), sind diese für die spätere Analyse relevant. Zu diesem Zweck wurden die Funktionen `x86_disable_intr`, `x86_enable_intr`, `trace_irqs_on`, `trace_irqs_off` und verwandte Funktionen analog zu den gewöhnli-

chen Sperrenoperationen modifiziert. Zur Identifizierung in der Nachverarbeitung wird als Speicheradresse ein speziell zugewiesener Pseudo-Wert genutzt.

6.1.4 Modifikationen des Speichermanagements

Neben dem Aufzeichnen der Sperrenoperationen ist der Mitschnitt der Speicherzugriffe imperativ, um die Korrektheit der Sperren prüfen zu können. Da es performancetechnisch in einem angemessenen Zeitrahmen nicht möglich ist, den kompletten Speicher des emulierten Systems zu beobachten und anschließend auszuwerten, wird dieses Aufzeichnen auf die zu analysierenden Datenstrukturen beschränkt. Ein weiterer Grund für diesen Ansatz besteht daraus, dass dies den Prozess des Zuweisen von Speicherbereich zu Datenstrukturen in der Nachverarbeitung wesentlich vereinfacht.

Implementiert ist dies durch das Einfügen von Aufrufen der `lockdoc_log_memory`-Funktion an relevanten Stellen. Das FAIL*-Framework ist bereits durch die vorangegangenen Experimente an Linux und FreeBSD in dieser Hinsicht modifiziert worden. Anders als bei der Instrumentierung der Sperren ist es an dieser Stelle nicht nötig, Informationen über die aufrufende Funktion (bspw. über `__FILE__`, `__LINE__` oder `__func__`) manuell aufzuzeichnen. Diese Daten werden während der Nachverarbeitung aus den entsprechenden Stacktraces ermittelt. Analog muss dieser Prozess für die jeweiligen Freigaben der beobachteten Datenstrukturen erfolgen.

Die eigentliche Methode kann ohne größeren Aufwand aus den bisherigen Arbeiten übernommen werden. Im Gegensatz zu sperrenbezogenenen Modifikationen ist es für die speicherbezogenen Modifikationen nicht nötig, allgemeine Änderungen vorzunehmen. Es muss lediglich innerhalb der relevanten Subsysteme instrumentiert werden. Während auch diese Informationen nicht relevant für die folgende Analyse sind, so dienen auch sie dem Abschätzen des Arbeitsaufwandes für zukünftige Arbeiten.

6.1.5 Modifikationen am LTP

Um das LTP in der Linux Emulationsschicht ausführen zu können bedarf es einiger Modifikationen. Die Version, welche in [LSBS19] und darauffolgenden Arbeiten genutzt wurde, Version 20190115, kann an dieser Stelle nicht direkt eingesetzt werden. Dies rührt daher, dass es nicht ohne weiteres möglich ist, LTP innerhalb der NetBSD Linux Emulationsschicht zu übersetzen. Abhilfe schafft das Übertragen der vorgenommenen Änderungen auf eine aktuellere Version, LTP 20230127. In dieser neueren Version

ist es möglich, LTP für `i386-linux-gnu` von einem `x86_64-linux-gnu` System aus zu kompilieren.

6.2 INSTRUMENTIERUNG

Der Erfolg von LockDoc ist abhängig davon, dass Speicherzugriffe auf die relevanten Datenstrukturen korrekt aufgezeichnet werden. Zu diesem Zweck muss jede Allokation von Speicher, welche anschließend von so einer Datenstruktur genutzt wird, identifiziert und an FAIL* weitergegeben werden.

Entgegen möglicher Erwartungen daran, dass diese relevanten Codestellen schwer aufzufinden sind oder es eine große Anzahl von ihnen gibt, ist dies nicht der Fall. Als Beispiel wird die Datenstruktur `struct vnode_impl` genutzt. Überprüfen der relevanten Code-Datei `sys/kern/vfs_vnode.c` zeigt lediglich zwei Code-Pfade, welche eine entsprechende Datenstruktur allozieren: `static vnode_impl_t * vcache_alloc(void)` und `vnode_t * vnalloc_marker(struct mount *mp)`. Eine genauere Übersicht über den stattgefundenen Arbeitsaufwand findet sich in Abschnitt 6.4.

6.2.1 Besonderheiten

Bei der Instrumentierung treten einige Sonderfälle auf, welche speziell behandelt werden müssen. Es folgt ein Auszug einiger dieser Besonderheiten.

6.2.1.1 Union Datentyp

Eine Datenstruktur, welche nicht anhand der LockDoc-Experimente abgebildet werden kann, ist der Union Datentyp. Eine Union stellt wie das Struct einen Verbunddatentyp dar, ist im Gegensatz zum Struct allerdings exklusiv bzgl. ihrer Mitglieder: Während mehrere innere Datenstrukturen existieren können, kann immer nur genau eine dieser Datenstrukturen einen Inhalt besitzen. Dementsprechend ist die Größe einer Union immer mindestens gleich der Größe der größten in ihr enthaltenen Datenstruktur.

In LockDoc kann dies zu Problemen führen, da dem Emulator zur Laufzeit nicht bekannt ist, welchen Datentyp die jeweilige Union an einem beliebigen Zeitpunk enthält. Abhilfe wird geschaffen, indem alle in zu beobachtenden Datenstrukturen enthaltenen Unions durch Structs ersetzt werden (vgl. Listing 5).

Durch diese Modifikation ergeben sich zwei Nachteile: Einerseits ergibt sich durch die dadurch entstehenden größeren Datenstrukturen ein erhöhter Speicherverbrauch

```
struct buf {
#ifndef LOCKDOC_VFS
        union {
#else
        struct {
#endif
#if defined(_KERNEL)
                /* LOCKDOC: This had to be moved in front of u_actq and
                 * u_rbnode becuase the vfs_bio subsys does
                 * some *interesting* pointer stuff
                 */
                struct work u_work;
#endif
                TAILQ_ENTRY(buf) u_actq;
                rb_node_t u_rbnode;
        } b_u;
        void        (*b_iodone)(struct buf *);
        int         b_error;
        // [...]
```

Listing 5: Beispiel zu Vermeidung von Unions in C, gekürzt, aus `sys/sys/buf.h`

des Kerns. Dieser ist allerdings so geringfügig, dass er innerhalb der stattgefundenen Experimente nicht messbar ist.

Außerdem kann es zu Komplikationen im Bezug auf an anderen Stellen stattfindende Zeigerarithmetik zu Problemen kommen. Wie in den Kommentaren in Listing 5 ersichtlich ist, musste im Zuge der Experimente die Reihenfolge der Mitglieder des Structs angepasst werden, da an anderer Stelle zum direkten Zugriff auf eine Instanz vom Typ `struct buf` ein Zeiger auf `struct work u_work` genutzt wird. Im unveränderten Falle der Unions ist dies kein Problem, da die Zeiger auf identische Speicheradressen verweisen. Wird die Union jedoch wie hier durch ein Struct ersetzt, so muss die Reihenfolge dementsprechend angepasst werden. Gäbe es nun mehrere solcher Fälle, also bspw. eine weitere Codestelle, welche auf einen Buffer anhand eines Zeigers zu `rb_node_t u_rbnode` zugreifen möchte, so hätten diese Stellen manuell angepasst werden müssen.

6.2.1.2 *Präprozessor Makros*

NetBSD macht, wie andere Betriebssysteme, starken Gebrauch von Präprozessor Makros, um Mitwirkenden einfachen Zugriff auf häufig genutzte Routinen zu bieten. Diese stellen eine Herausforderung für LockDoc dar, da zum Zeitpunkt der Experimentausführung oder der Analyse keine Informationen mehr darüber vorhanden sind, welchen Namen ein Code-Block ursprünglich hatte. Dieser kann somit weder gefiltert werden, noch kann dieser Name in menschlich lesbaren Ausgaben genutzt werden.

Ein Beispiel hierfür sind auch Kernel-Funktionen, welche atomaren Zugriff auf Datenstrukturen zur Verfügung stellen, sodass in einigen Fällen kein explizites Locking notwenig ist. Somit lässt sich für NetBSD die Funktion `atomic_load_relaxed` nennen, welche als C Makro implementiert ist (siehe Listing 6) und daher während der Kompilierung durch die aufgelisteten internen Funktionen ersetzt wird. Dies macht es nicht möglich, sie der Filterliste hinzuzufügen.

6.2.2 *Filterung*

Wie in Abschnitt 3.1 bereits kurz beschrieben, ist es zum effektiven Einsatz von LockDoc nötig eine Liste von Funktionen aufzustellen, welche von der eigentlichen Analyse ausgeschlossen werden sollten. Während die Länge dieser Liste in anderen Betriebssystemen stark ansteigen kann, so ist sie in NetBSD kurz genug, als dass sie im Folgenden vollständig erläutert werden kann.

```
#define atomic_load_relaxed(p)                                  \
({                                                              \
    const volatile __typeof__(*(p)) *__al_ptr = (p);            \
    __ATOMIC_PTR_CHECK(__al_ptr);                               \
    __BEGIN_ATOMIC_LOAD(__al_ptr, __al_val);                    \
    __END_ATOMIC_LOAD(__al_val);                                \
})
```

Listing 6: Implementierung von `atomic_load_relaxed`, aus NetBSD `sys/sys/atomic.h`.

Die genutzte Filterliste lässt sich in zwei Hälften aufteilen. Die erste Hälfte besteht vollständig aus atomaren Operationen, da diese wie in Abschnitt 5.1 beschrieben nicht auf Sperren jeglicher Art angewiesen sind, NetBSD-spezifisch also jegliche atomare Operationen, welche in `man 3 atomic_ops` gelistet sind.

Die zweite Hälfte besteht aus den Funktionen, welche eine der zu beobachtenden Datenstrukturen initialisieren. Für `struct vnode_impl` und dessen verschachtelten Elemente ergeben sich somit `vcache_new`, `vcache_get`, `vcache_reclaim` und `vrelel`, welche jeweils zum Erstellen neuer, Laden existierender oder Freigeben nicht mehr benötigter Vnodes genutzt werden. Aus diesem Grund finden in diesen Routinen oftmals Zugriffe auf Datenstrukturen statt, für welche keine angemessenen Sperren geholt wurden. Bei den Funktionen zum Erstellen bzw. Laden von Vnodes ist dies ungefährlich, da zu dem Zeitpunkt der Speicher neu alloziert wurde und somit keine anderen Prozesse von dessen Existenz wissen bzw. Zeiger zu ihnen besitzen. Die jeweiligen Funktionen zum Freigeben deallozieren diesen Speicher am Ende des Lebenszyklus eines Vnodes wieder. Zuvor stellen diese über sperrenunabhängige Mechanismen sicher, dass keine Referenzen mehr zu der Instanz existieren und dass diese über einen Enum als unbenutzbar markiert ist.

Eine vergleichbare Situation ergibt sich für `struct buf`, hier werden die Funktionen `genfs_do_io` und `genfs_getpages_read` gefiltert. Beide Routinen nutzen `getiobuf`, um eine Referenz auf leere Datenstruktur aus dem Pool zu erhalten.

Es ist anzumerken, dass der Übersicht halber die Filterlisten nur Funktionen beinhalten, welche oft genutzt werden, da diese sonst die Auswahl einer Hypothese beeinflussen. Routinen, welche nur selten oder einmalig aufgerufen werden, können zum späteren Zeitpunkt in der manuellen Nachverarbeitung geprüft werden.

6.3 HERAUSFORDERUNGEN

Während der Anpassung von NetBSD zum Einsatz in den LockDoc-Experimenten ergeben sich eine Reihe an Herausforderungen. Während einige problematischen Stellen im Bezug auf die Instrumentierung bereits genannt wurden, so werden nun allgemeine, konzeptuelle Herausforderungen aufgeführt.

6.3.1 *Synchronisation ohne explizite Sperren-Datenstrukturen*

Einige Datenstrukturen nutzen keine Sperren im klassischen Sinne zur Gewährleistung von korrekter Synchronisierung. Ein Beispiel findet sich in der Datenstruktur `struct buf`. Dieses nutzt das Mitglied `u_int b_cflags` als Flag Array. Wird dort ein gewisses Bit gesetzt, so verdeutlicht dies, dass das umliegende Struct für weitere Zugriffe gesperrt ist.

Da das setzen dieser Flag i. d. R. nicht[5] über einen zusätzlichen Funktionsaufruf geschieht, ist es nicht möglich, diesen zum Mitschneiden der Operationen so neuzudefinieren wie es mit anderen Sperrenoperationen geschehen ist. Stattdessen muss jeder Zugriff auf diese Flag manuell instrumentiert werden. Dieser Prozess wird dadurch erleichtert, dass in NetBSD diese Flags über C Makros spezifiziert werden, in diesem Fall handelt es sich um `#define BC_BUSY 0x00000010`. Eine simple Codesuche nach `BC_BUSY` ergibt etwa 50 solche Instanzen, in denen diese Sperre bearbeitet wird. An solchen Stellen muss manuell instrumentiert werden, ein Beispiel findet sich in Listing 7.

```
bp->b_cflags |= BC_BUSY;
#ifdef LOCKDOC
        lockdoc_log_lock(P_WRITE, &(bp->b_cflags), __FILE__, __LINE__,
        ↪  __func__, "b_cflags", 0);
#endif
```

Listing 7: Beispiel zur zusätzliche Instrumentierung von Sperren durch Flags.

[5] Es existieren Makros zum setzen/entfernen dieser Flag, diese Makros werden allerdings nur selten genutzt.

Außerdem müssen die Werkzeuge zur Nachverarbeitung so angepasst werden, dass zusätzlich zu klassischen Mutexen auch die neue Art von Sperre, genannt `b_cflags`, korrekt analysiert wird.

6.3.2 *Sperren außerhalb von Structs*

An verschiedenen Codestellen tritt der Fall auf, dass eine zu beobachtende Datenstruktur und die darin enthaltenen Mitglieder zwar zugewiesene Sperren besitzen, diese Sperren aber weder zum Zeitpunkt der Kompilierung alloziert werden, sie sich aber auch nicht in dem Struct selbst oder in einer bereits anderweitig beobachteten Datenstruktur, wie es der Fall bei verschachtelten Structs ist, befinden.

Ein solches Beispiel findet sich in `struct vnode_impl` und `struct vnode`. Dort werden einige Mitglieder (laut Dokumentation und Testergebnissen) durch `v_interlock` gesperrt. Dieses befindet sich jedoch in `struct vnode` und besteht lediglich aus ein Zeiger zum entsprechenden Lock, welches parallel zum `vnode` initialisiert wird.

Um dieses Problem zu umgehen oder zu lösen wurden mehrere Methoden evaluiert, hier erläutert anhand des Beispiels `kmutex_t *v_interlock` aus `struct vnode` (vgl. Listing 4).

6.3.2.1 *Auffinden des referenzierten Speicherbereiches*

Eine Musterlösung für dieses Problem, welche bereits in Abschnitt 3.2 beschrieben wurde, bestünde aus der rekursiven Suche nach Zeigern in den zu beobachtenden Datenstrukturen seitens FAIL*. Dieses Problem weist Ähnlichkeiten zu Algorithmen zur Garbage Collection auf, mit dem wichtigen Unterschied, dass der Algorithmus keine Informationen über die Datenstruktur selbst (insb. deren Größe) hat, sondern lediglich den Speicherinhalt sieht. Diese Tatsache macht es schwer Möglich, eine rekursive Suche nach zu beobachtenden Speicherbereichen zu implementieren.

6.3.2.2 *Manuelles beobachten des Speicherbereichs der Sperre*

Ein weiterer Lösungsansatz, welcher bei der Entwicklung dieser Arbeit evaluiert wurde, besteht aus dem manuellen Hinzufügen dieser problematischen Sperren zu dem zu beobachtenden Speicherbereich. Am Beispiel von `struct vnode_impl` und `v_interlock` würde dies in dem in Listing 8 abgebildeten Code resultieren.

```
#ifdef LOCKDOC
lockdoc_log_memory(1, "vnode_impl", vip, sizeof(*vip));
lockdoc_log_memory(1, "kmutex_t", vp->v_interlock,
↪  sizeof(*(vp->v_interlock)));
#endif
```

Listing 8: Beispiel für manuelles Beobachten der Speicherbereiche der problematischen Sperren

Da LockDoc in der Analyse-Phase für Operationen auf allen Sperren überprüft, ob diese entweder eine globale, statische Sperre ist oder eine dynamisch allozierte Sperre in einem beobachteten Speicherbereiche ist (vgl. Abschnitt 3.2), löst das manuelle Hinzufügen dieser Speicherbereiche augenscheinlich das beschriebene Problem auf eine simple Art und Weise.

Der Implementierungsaufwand dieser Behelfslösung ist überschaubar, da es i. d. R. eine einstellige Anzahl an Codepfaden gibt, die eine bestimmte Datenstruktur allozieren.

Dieser Lösungsweg führt allerdings in der späteren Auswertung und Nachverarbeitung zu beachtlichen Problemen: Zu diesem Zeitpunkt kann nicht mehr festgestellt werden, welcher zusätzlich beobachteter Mutex (oder welche Sperre generell) zu welcher Datenstruktur gehört. In der manuell zu sichtenden Ausgabe werden bspw. sämtliche Mutexe als `EMBOTHER(kmutex_t.mtxa_owner-s[w])` aufgelistet.

Dies ist allerdings schon vor der manuellen Auswertung ein wichtiges Problem. Als Beispiel wird eine gewöhnliche Schreiboperation auf ein beliebige Datei betrachtet. In so einem Fall wird i. d. R. auf die entsprechende Vnode Datenstruktur und einen Buffer zugegriffen. Es ist also nicht unwahrscheinlich,[6] dass zeitnah zueinander auf die folgenden Variablen zugegriffen wird:

- `vnode.v_usecount` (Synchronisation über `kmutex_t *v_interlock`)
- `buf.b_oflags` (Synchronisation über `kmutex_t *b_oflag`)

Beide Variablen werden lediglich über einen Zeiger auf einen Mutex synchronisiert. Genauere Informationen über die Instanz des Mutexes oder dessen Zugehörigkeit gehen bei dem vorgestellten Lösungsversuch allerdings verloren. Ist nun eine dieser

[6] Dieses Verhalten konnte in einigen Experimenten belegt werden.

beiden Datenstrukturen nicht korrekt gesperrt, so ist es LockDoc nicht möglich, dies zu erkennen: Fehlt eine Sperre für den Zugriff auf das Vnode, so kann die für den Buffer gesetzte Sperre fälschlicherweise als solche interpretiert werden. Selbiges gilt auch umgekehrt.

Dies ist insb. problematisch, da es in NetBSD in den beobachteten Datenstrukturen mehrere Mitglieder gibt, welche von dieser Taktik der Zeiger auf Sperren Gebrauch machen.

Alleine diese Problematik macht den vorgestellten Lösungsansatz bereits unbrauchbar für diese Arbeit. Es existieren jedoch noch weitere konzeptuelle Probleme dieser Lösung: Speicherbereiche, welche bei der Initialisierung der Datenstrukturen an dem Emulator übermittelt werden, sind nicht zwingend konstant. Für das Struct `struct uvm_object`, welches direkt in `struct vnode` eingebettet ist, existiert das Mitglied `struct krwlock * vmobjlock`. Diese Sperre wird für einen Teil der Mitglieder des Vnodes genutzt (vgl. Anmerkung „u“ in Listing 4). Der beschriebene Lösungsvorschlag lässt sich allerdings nicht ohne weiteres auf diese Sperren anwenden, da Funktionen wie `void uvm_obj_setlock(struct uvm_object *uo, krwlock_t *lockptr)` existieren, welche diese Sperre durch eine andere Instanz eines RWLocks ersetzen. Geschieht dies, so ist nicht mehr abzusehen, welche Daten der bisherig beobachtete Speicher enthalten wird, außerdem werden von diesem Zeitpunkt an Zugriffe auf die Sperre in der Analyse verworfen, da sie nun außerhalb der beobachteten Speicherbereiche liegt.

6.3.2.3 Manuelles Beobachten der relevanten Sperrenoperationen

Eine weitere Methode dieses Problem zu umgehen besteht aus dem manuellen instrumentieren des Codes so, dass Operationen auf den problematischen Sperren direkt an den Emulator weitergeleitet werden. Dies geschieht auf die selbe Weise, wie die erweiterten Funktionen zum Aufzeichnen gewöhnlicher Sperren dies tun.

So müsste jedes Sperren bzw. Entsperren eines der Problematischen Locks mit zusätzlicher Instrumentierung versehen werden, sodass diese korrekt aufgezeichnet werden. Ein Beispiel könnte wie in Listing 9 beschrieben aussehen.

Wie aus diesem Beispiel bereits ersichtlich wird, ist dies mit einem wesentlich höheren Arbeitsaufwand verbunden, da sämtliche Aufrufe von Sperrenoperationen auf den Sperren identifiziert und instrumentiert werden müssen. In dem vorgestellten Anwendungsbeispiel auf das VFS muss für diesen Lösungsansatz jedes einzelne Dateisystem manuell aufwändig zur Aufzeichnung der Sperren instrumentiert werden. Des Weiteren

```
struct vnode *vp = // [...];

#ifdef LOCKDOC
lockdoc_log_lock(P_WRITE, PSEUDOLOCK_ADDR, __FILE__, __LINE__, __func__,
↪  "kmutex_t", 0);
#endif
mutex_enter(vp->v_interlock);

// Datenzugriff auf Mitglieder von vp

#ifdef LOCKDOC
lockdoc_log_lock(V_WRITE, PSEUDOLOCK_ADDR, __FILE__, __LINE__, __func__,
↪  "kmutex_t", 0);
#endif
mutex_exit(vp->v_interlock);
```

Listing 9: Beispiel für manuelles Beobachten der Sperrenoperationen auf problematischen Sperren

ist dieser Prozess durch diese erweiterte manuelle Arbeit auch fehleranfälliger als etwaige Alternativen.

Ein weiteres Problem dieser Lösung ergibt sich dadurch, dass es nötig ist die Adresse der Sperre zu modifizieren, damit die nachgelagerten Analysewerkzeuge diese aufgrund nicht bekannter Speicherbereiche nicht direkt wieder verwerfen. In dem oben aufgeführten Beispiel wird als Speicheradresse die speziell dafür angelegte Pseudoadresse `PSEUDOLOCK_ADDR` genutzt, welche in der anschließenden Weiterverarbeitung als solche erkannt wird und entsprechend behandelt wird. Es ist allerdings zu beachten, dass die Analysewerkzeuge die Adresse der Sperre als deren Identifikationsmerkmal nutzen. Wird diese also durch eine Pseudoadresse ersetzt, so kann nicht mehr zwischen unterschiedlichen Instanzen der selben Art von Sperre unterschieden werden. Dies führt zu wiederholten doppelten Sperren bzw. Entsperren des augenscheinlich gleichen Locks. Zudem entstehen dadurch, dass Sperren hierdurch ambivalent werden die selbe Problematik der nicht mit Datenstrukturen assoziierbaren Sperren wie im vorherigen Lösungsversuch (vgl. Unterunterabschnitt 6.3.2.2).

Aus den bisherigen Überlegungen zur geeigneten Behandlung dieser problematischen Zeiger auf Sperren wird ersichtlich, dass ein gesondertes Verfahren, welches im Kernel selbst implementiert werden muss, ungeeignet scheint. Daher wurden auch Überlegungen angestellt und evaluiert, wie sich dieser Umstand in die Werkzeuge zur Nachverarbeitung integrieren lässt.

Zu diesem Zweck muss die Person, welche LockDoc anwenden möchte, statt der bisherigen, aufwändigen manuellen Instrumentierung direkt im Kernel lediglich die Namen der Zeiger, welche diese Sperren in ihren jeweilig zugehörigen Structs haben, in eine Liste eintragen. Diese Liste wird während der Analyse eingelesen. Für jede zu verarbeitende Sperre wird überprüft, ob der Name der Variable, in der der Zeiger auf die Sperre gespeichert ist, einem der problematischen Locks gleicht.

Der Name der Variable wird durch ein C Präprozessor Makro während der Übersetzung des Kernels in den Funktionsaufruf zum Mitschneiden der Sperre injiziert.

Wird so eine Sperre gefunden, muss entschieden werden, wie mit solchen Sonderfällen weiter umzugehen ist. Auch hier wurden mehrere Ansätze betrachtet.

6.3.2.4 Behandeln der Sperren als statische Sperren

Als erste Methode zum Problembehandlung wurde das Behandeln dieser Sperren analog zu den globalen, statischen Sperren geprüft. Dies ermöglicht zwar eine simple Implementierung (dem Setzen der zur Sperre gehörigen Allokations-ID auf 0), führt allerdings zu einem erheblichen Problem: Sobald mehr als eine Instanz einer solchen Sperre existiert, ergibt sich (wie bei der manuellen Sperreninstrumentierung über eine Pseudoadresse in Unterunterabschnitt 6.3.2.2) die selbe Problematik wie zuvor: Es kann nicht mehr zwischen unterschiedlichen Instanzen unterschieden werden. Dies resultiert neben Fehlermeldungen über doppeltes Sperren und Entsperren in dem bekannten Problem, dass Sperren nicht mehr ihren Datenstrukturen zugewiesen werden können. Dieser Umstand macht diese Methode dementsprechend ungeeignet als Lösung des vorhandenen Problems.

6.3.2.5 Behandeln der Sperren als globale Pseudoallokation

Innerhalb der Analysewerkzeuge existieren neben der Allokations-ID 0 für statische Datenstrukturen noch weitere spezielle Allokations-IDs. Eine solche ist die Pseudo-Allokations-ID. Wird eine problematische Sperre erkannt und ihre Allokations-ID

entsprechend angepasst, so ergeben sich jedoch in der Weiterverarbeitung der Daten die selben Probleme wie im vorherigen Fall: Auch hier kann nicht mehr zwischen unterschiedlichen Instanzen der selben Sperre unterschieden werden.

6.3.2.6 Behandeln der Sperren als eine Instanz

Um die bisherigen Probleme zu umgehen, wurde eine Methode betrachtet, welche beim Einlesen der generierten Trace-Dateien die problematischen Sperren über den beschriebenen Weg identifiziert und auch anhand ihrer korrekten Speicheradressen identifiziert, allerdings eine modifizierte (identische) Speicheradresse zur nachgelagerten Hypothesenaufstellung und Fehlersuche in die Datenbank schriebt. Dies stellt sich als ungeeignet heraus, da somit das logische Datenbankschema verletzt wird. Außerdem ist es dem Hypothesizer so nicht möglich, aus den Speicheradressen korrekt rückwirkend auf die Namen bzw. Instanzen der jeweiligen Sperren zu schließen.

6.3.2.7 Behandeln der Sperren als Pseudoallokation je Instanz

Die letzte evaluierte Methode besteht aus einer größeren Erweiterung der Werkzeuge zur Nachverarbeitung der Daten. Zunächst werden wie in den bisherigen Ideen problematische Sperren anhand ihrer Namen innerhalb der Structs erkannt. Diese Liste der Sperren wurde durch den Datentyp der Sperre und deren Größe erweitert. Zum Startpunkt der Nachverarbeitung wird diese Liste eingelesen. Aus ihr werden (zwecks späterer Ereignisbehandlung) neue Structs halluziniert, welche die zuvor spezifizierte Größe besitzen. Da Structs in C nicht mehr Speicher als die in ihnen enthaltenen Datenstrukturen einnehmen,[7] können Größe von Struct und Sperre gleichgesetzt werden. Eine Veranschaulichung findet sich in Listing 10.

Wird nun eine der Sperrenoperationen eingelesen, deren Sperrenname sich innerhalb der Liste wiederfindet und die bisher noch nicht bekannt ist, so wird diese im weiteren Verlauf so interpretiert, als würde es sich um eine neue Instanz des halluzinierten Structs handeln. Weitere Operationen auf der Sperre (identifiziert anhand ihrer Adresse) werden auch diesem Struct zugewiesen.

[7] Eine Ausnahme besteht aus dem Padding von Mitgliedern, dies ist hier dadurch, dass nur ein Mitglied existiert, nicht der Fall.

```
v_interlock;kmutex_t;8
```

```
struct v_interlock {
        kmutex_t v_interlock;
}
```

Listing 10: Ein- und (virtuelle) Ausgabe des verbesserten Ansatzes zur Behandlung von Zeigern auf Sperren. Es ist anzumerken, dass die Ausgabe aufgrund der doppelten Definition kein korrekter C Code ist.

Ein Nachteil dieser Methode besteht daraus, dass nicht erkannt werden kann, wann Sperren freigegeben werden – somit ist es nicht möglich, den Speicher, der von den virtuellen Structs eingenommen wird wieder freizugeben. In der Theorie kann dies zu Problemen führen, wenn dieser Speicher von einer anderen zu beobachteten Datenstruktur wiederverwendet wird. Dieser Fall ist relativ unwahrscheinlich und ist in keinem der im Rahmen dieser Arbeit stattgefundenen Experimenten je aufgetreten.

Das Portieren von LockDoc und das Instrumentieren der relevanten Subsysteme ist an dieser Stelle abgeschlossen. Im Zuge dessen wurden einige Probleme dieses Prozesses analysiert und zum Teil gelöst oder umgangen. Es folgt nun eine Übersicht über den nötigen Arbeitsaufwand für etwaige Portierungen von LockDoc auf weitere Betriebssysteme.

Anschließend werden mit Hilfe des instrumentierten NetBSDs in Kapitel 7 die eigentlichen Experimente ausgeführt und anschließend deren Ergebnisse evaluiert.

6.4 AUFWAND

Ein hoher Implementierungsaufwand steht der zukünftigen Adaption des LockDoc-Ansatzes entgegen, da dieser mögliche Kernel-Mitwirkende von dessen Umsetzung abschrecken kann. Ziel von LockDoc ist es daher diesen möglichst gering zu halten. An dieser Stelle wird analysiert, wie hoch dieser Aufwand bei der Portierung, Implementierung und Instrumentierung zu/in NetBSD ist.

Hier wird nur der Portierungsaufwand für die ursprüngliche LockDoc-Methode betrachtet. Etwaiger zusätzlicher Code, welcher aufgrund der Erweiterung von LockDoc

Datenstruktur	Allokation und Deallokation	Sonderfälle	Gesamt
`struct vnode_impl`	4	0	0
`struct vnode`	0	4	4
`struct mount`	2	0	2
`struct buf`	2	50	52
Total	8	54	62

Tabelle 6.4.1: Anzahl der zusätzlichen Zeilen Code, welche zum Instrumentieren (Summe aus Allokation und Deallokation) der jeweiligen Datenstruktur, sowie zur Behandlung von Sonderfällen notwendig ist. Zusätzlich zu dieser Anzahl ist es empfehlenswert, je zwei Zeilen für etwaige `#ifdef`-Statements hinzuzurechnen.

selbst oder zur Behandlung von NetBSD-spezifischen Problemen geschrieben werden musste, wird an dieser Stelle nicht mit einberechnet.

Die notwendigen Veränderungen lassen sich auf vier Kategorien abbilden:

6.4.1 Portierung

Das Portieren der notwendigen Methoden zur Kommunikation zwischen Betriebssystemkernel und FAIL* besteht aus mehreren Elementen.

Zuerst müssen die benötigten Funktionen in den Kernel integriert werden. Dies besteht aus dem portieren der entsprechenden C-Header und -Dateien (hier `sys/sys/lockdoc.h` und `sys/lockdoc/log.c`) und ggf. dem Anpassen des Build-Systems, sodass diese bei zukünftigen Kompilierungen mit integriert werden. Unter NetBSD ist dieser Aufwand nahezu nicht vorhanden: Da Betriebssystemkerne i. d. R. in C implementiert sind und der LockDoc-Code keine externe Abhängigkeiten besitzt, kann das bereits in FreeBSD genutzte Code-Skelett (bis auf einige wenige Umbenennungen von Funktionen aufgrund von Konflikten) problemlos übertragen werden.

Wie bereits in Unterabschnitt 6.1.1 beschrieben ist eine minimale Kernelkonfiguration wünschenswert. Eine solche Konfiguration zu erstellen, welche sowohl alle nicht benötigten Kernel-Module deaktiviert und die entsprechenden LockDoc-Elemente

aktiviert, ist bei den meisten Betriebssystemen – somit auch bei NetBSD – trivial, i. d. R. sind diese Konfigurationsdateien gut dokumentiert. Die Tatsache, dass für die Ausführung in FAIL* bzw. Bochs lediglich sehr primitive Module und PC-kompatible Treiber benötigt werden, vereinfacht diesen Prozess erheblich.

Zuletzt muss das Übertragen von Laufzeitinformationen (vgl. Unterabschnitt 6.1.2) implementiert werden. In der NetBSD-Portierung wurden diese in `sys/lockdoc/init_data.c` implementiert. Auch hier ist ein einfaches Übertragen des bereits existierenden C-Codes möglich, lediglich einige interne Namen von Datentypen mussten modifiziert und das Aufrufen der jeweiligen Funktionen beim Starten des Betriebssystem (4 Zeilen Code und zugehörige `#ifdef`-Statements in `sys/kern/init_main.c`) zusätzlich implementiert werden.

Es ist zu beachten, dass all diese hier genannten Modifikationen einmalig sind und somit nicht bei dem Miteinschließen von weiteren Datentypen, Subsystemen oder Sperren wiederholt werden müssen

6.4.2 *Speicherbeobachtung*

Eine Übersicht über die benötigte Anzahl an zusätzlichen Zeilen Code findet sich in Tabelle 6.4.1. Für einige Datentypen ist für die Instrumentierung der Allokation und Deallokation ein Aufwand von 0 Zeilen Code angegeben. Diese Datenstrukturen müssen nicht gesondert instrumentiert werden, da sie ausschließlich in andere, bereits beobachtete Datenstrukturen eingebettet sind.

Insbesondere im Vergleich zu anderen Analysemethoden wie statischer Analyse, welche Größenordnungen von 20 Zeilen Annotationen pro 1.000 Zeilen Code nennen (vgl. Unterabschnitt 2.3.3), ist für die Untersuchung mittels LockDoc nur ein Bruchteil dieses Aufwandes erforderlich.

6.4.3 *Sperrenbeobachtung*

Das Implementieren des notwendigen Skeletts zur Sperrenbeobachtung ist zwar einer der am längsten andauerndsten Aufgaben bei der Portierung, allerdings auch einer der einfachsten. Hierfür müssen zuerst die relevanten Arten von Sperren ausfindig gemacht werden (vgl. Abschnitt 5.1). Ist dies geschehen, so können in `sys/sys/lockdoc.h` entsprechende `#define` Makros eingefügt/abgeändert werden, welche die existierenden Methoden zum Sperren/Entsperren zu neuen, aufzeichnenden Methoden umschreibt.

Diese neuen, aufzeichnenden Methoden müssen dementsprechend auch implementiert werden, sie rufen allerdings lediglich `lockdoc_log_lock` auf, gefolgt von der ursprünglichen Funktion. Insgesamt ergibt sich so für 10 relevante Sperrenmethoden und 6 weitere Methoden, welche für die Unterbrechungsbehandlung verantwortlich sind ein Arbeitsaufwand von 26 Zeilen Code im LockDoc Kernel Modul selbst, gefolgt von etwa 15 weiteren Zeilen Code in den jeweiligen Headern der eigentlichen Sperren. Insgesamt entsteht so ein Portierungsaufwand von lediglich unter 100 Zeilen Code zum beobachten der Sperren.

An dieser Stelle ist jedoch ein zusätzlicher Portierungsaufwand nennenswert: Dadurch, dass in fortschrittlichen Betriebssystemen einige Sperrenoperationen aus Performanzgründen direkt in Assembly implementiert sind, müssen jene entweder zusätzlich instrumentiert oder gar deaktiviert werden. Da dieser Aufwand stark abhängig vom jeweiligen Kernel ist und unter NetBSD durch das Vorhandensein des LockDebug-Frameworks vereinfacht wurde (vgl. Unterunterabschnitt 6.1.3.1), lässt sich der Aufwand an dieser Stelle nicht pauschalisieren.

Auch hier gilt, dass all diese genannten Modifikationen, wie jene zur eigentlichen LockDoc-Portierung, einmalig sind, und damit nicht für neue Subsysteme oder Datenstrukturen wiederholt werden müssen.

6.4.4 *LockDoc Werkzeuge*

Innerhalb der LockDoc Werkzeuge ist es kaum nötig, betriebssystemabhängige Modifikationen zu betreiben. Lediglich das Hinzufügen der beobachteten Sperrentypen (hier `kmutex_t` und `krwlock_t`) zum Konvertierungsskript sind nötig.

Es ergibt sich somit ein vergleichsweise geringer Arbeitsaufwand und damit eine geringe Hemmschwelle zur Implementierung der LockDoc-Experimente auf neuen Betriebssystemen.

6.5 ZUSAMMENFASSUNG

Es konnte gezeigt werden, wie LockDoc auf ein bisher nicht unterstütztes Betriebssystem portiert werden kann, einschließlich der dabei auftretenden Probleme und deren Lösungen. Zusätzlich wurde beschrieben, wie relevante Datenstrukturen instrumentiert werden können und welche Sonderfälle dabei zu beachten sind.

Zudem wurde begutachtet, wie groß der benötigte Arbeitsaufwand insgesamt ist und inwiefern dieser auch auf zukünftige Portierungen zutrifft.

Als Nächstes folgt die Ausführung der so vorbereiteten Experimente, die daraus gewonnen Daten werden in den anschließenden Abschnitten genauer analysiert.

7 EVALUATION

Nachdem das zu untersuchende Betriebssystem erfolgreich instrumentiert worden ist, ist es möglich, die gewählten Arbeitslasten in der Testumgebung auszuführen und die so gewonnen Daten zu analysieren.

Rückblickend auf das vorgestellte Flussdiagramm in Abbildung 5.4.1 wurden die Komponenten *Workload*, *Operating System*, *FAIL*/Monitoring* und *Post Processing* erfolgreich erweitert. Es folgt in diesem Kapitel die Ausführung und Modifizierung (vgl. Unterunterabschnitt 7.3.1.2) des *Locking-Rule Derivator*.

Anhand der daraus resultierenden Ergebnisse wird das Betriebssystem in Abschnitt 7.5 auf *Lockung-Rule Specification Bugs* hin untersucht. Dazu benötigt es das vorherige Übersetzen der *Documented Locking Rules* in ein maschinenlesbares Format, einige Besonderheiten dieses Vorgehens sind in Unterabschnitt 7.5.1 erläutert.

7.1 EVALUATION DER KORREKTHEIT DER MODIFIKATIONEN

Um diese stattgefundenen Modifikationen auf ihre Funktionsweise und insb. Richtigkeit zu Überprüfen, wurde sich der von Lochmann *et. al.* vorgestellte LockDoc-Test zunutze gemacht. Dieser wurde analog zu seinen Linux- und FreeBSD-Implementierungen auch in NetBSD durch ein zusätzliches Kernel-Modul verwirklicht. Zweck dieses Modules ist es, Datenstrukturen zu erstellen und so zu nutzen, dass diese vor einer festgelegten Anzahl an Zugriffen korrekt gesperrt sind. Anschließend finden Zugriffe auf diese Datenstrukturen statt, welche nicht adäquat gesperrt sind. Alle stattfindenden Sperren- und Speicheroperationen werden durch die zuvor implementierten Methoden zum Aufzeichnen von Sperren- und Speicherzugriffen mitgeschnitten. Nach Abschluss dieses Durchlaufens werden die resultierenden Aufzeichnungen in die Pipeline zur Nachverarbeitung gegeben. Ist jeder vorangegangene Schritt (Speicher- und Sperreninstrumentierung, Emulator, Post-Processing-Pipeline) korrekt implementiert, so zeigt die finale Ausgabe dieses Tests die absichtlich ungesperrten Zugriffe auf die Test-

Datenstruktur an. Ist dies der Fall, so sind die bisherigen Modifikationen dazu geeignet, den eigentlichen Kernel zu überprüfen.

7.2 VERSUCHSAUFBAU

Zum Zweck der späteren Ausführung von Experimenten in FAIL* wurde zunächst ein minimales NetBSD 10.99.5 Systems erstellt. Nach der dortigen Installation der benötigten Abhängigkeiten der Benchmarks wurden der modifizierte Kernel (kompiliert mit der LOCKDOC-Konfiguration), die angepassten Bootloader-Konfigurationen und die erweiterten Initialisierungs-Skripte auf das System übertragen.

Die Experimente wurden auf einem Linux System ausgeführt. Unter Nutzung des FAIL*-Emulators in Verbindung mit den Verschiedenen Arbeitslasten ergibt sich eine Laufzeit von etwa 5 Tagen und 17 Stunden für die FS Testsuite von ATF und eine Laufzeit von 2 Tagen und 4 Stunden für die FS Testsuite von LTP. Diese Experimente fanden auf einem Intel Core i7-12700 statt. Die eigentliche Analyse der Daten geschah hingegen auf vier Intel Xeon E5-4640. Eine genauere Aufschlüsselung des benötigten Zeit- und Speicheraufwandes findet sich in Tabelle 7.2.1.

7.3 STRATEGIE

Bei der Analyse des zuvor gewonnenen Datensatzes stehen die in Unterabschnitt 2.4.4 beschriebenen Algorithmen zur Auswahl von Hypothesen zur Verfügung. An dieser Stelle kommt nun die Frage auf, welche dieser Strategien am besten dazu geeignet ist, sinnvolle Hypothesen aufzustellen, welche sich zur Erkennung von Implementierungsfehlern eignen. Zunächst findet also eine Art Benchmark der durch verschiedene Algorithmen gewonnenen Hypothesen statt. Als Vergleichswert dient hierfür eine Grundwahrheit, welche aus der Dokumentation der zu beobachtenden Datenstrukturen erstellt wurde.

Es wurden die vier Algorithmen (Top Down, Bottom Up, Sharpen und LockSet) auf jeweils zwei Datensätzen evaluiert, welche durch Durchlaufen der ATF File System Test Suite und der LTP File System Test Suite entstanden sind. Bei dieser Evaluation wurden jeweils unterschiedliche Parameter pro Algorithmus gewählt, so sind in bei Top Down und Bottom Up der jeweilige Schwellwert zum Auswahl einer Hypothese t_{ac} variabel, beim Sharpen-Algorithmus ist der Schwellwert, um den sich die Supportrate beim Hinzufügen einer Sperre verschlechtern darf, t_s, variabel (vgl. Unterunterabschnitt 2.4.4.1).

Benchmark	Größe Datensatz	Anzahl Ereignisse	Laufzeit Experiment	Laufzeit Analyse	Laufzeit Gesamt
LockDoc-Test	276,68 MiB	1.845.935	35 min	17,28 s	35 min
ATF FS Test Suite	206,25 GiB	1.205.048.730	137 h	11,57 h	≈ 149 h
LTP FS Test Suite	108,45 GiB	594.202.611	52 h	10,75 h	≈ 63 h
Total	≈ 316 GiB	1.901.097.276	≈ 190 h	≈ 22 h	≈ 212 h

Tabelle 7.2.1: Informationen über den benötigten Zeit- und Speicheraufwand der verschiedenen Arbeitslasten unter NetBSD. Die Größe der Datensätze wurde anhand der Dateigröße der von FAIL* generierten CSV-Dateien gemessen.

Der LockSet Algorithmus bietet an sich keine solchen Parameter. Gemessen wird die so erzielte Leistung der unterschiedlichen Algorithmen bzw. Parameter anhand des prozentualen Anteils an Sperren-Regeln, welche jeweils identisch zu denen der Grundwahrheit sind.

Die Ergebnisse finden sich in Abbildung 7.3.1 und Abbildung 7.3.2. Diese Ergebnisse wurden durch Auswahl des aggregierten (NoWoR), Kontext-sensitiven (CTX) Ansatzes erzielt (vgl. Unterunterabschnitt 2.4.4.2 und Unterunterabschnitt 2.4.4.3). Es werden zunächst die Daten der strikten Prüfung betrachtet, diese entspricht den genutzten Evaluationsalgorithmen aus vorherigen Arbeiten von Lochmann *et. al.*

Im Allgemeinen wird durch diese Graphen ersichtlich, dass über den Top Down-Algorithmus die höchste Supportrate in beiden Datensätzen erzielt werden kann. Diese sinkt wie zu erwarten mit steigendem Schwellwert, da ein höherer Schwellwert in diesem Kontext gleichzusetzen mit einer geringeren Toleranz gegenüber möglichen Implementierungsfehlern oder fehlenden Filterlisteneinträgen ist.

Zum einen fällt auf, dass insb. im Vergleich zu vorherigen Arbeiten (siehe bspw. [Loc21, Abb. 7.5]) sowohl durch den Einsatz von Bottom Up, als auch von Sharpen als Strategie zur Hypothesenauswahl ein relativ abrupter Anstieg der übereinstimmenden Sperren-Regeln bei einem strikter gewählten Schwellwert ($t_{ac} \geqslant 98\%$ bzw. $t_s \leqslant 0{,}2$) geschieht. Durch diese ansteigende „Korrektheit“ der Hypothesen als Folge der Verminderung für Fehlertoleranz lässt sich daraus schließen, dass das NetBSD Projekt sein Versprechen der hohe Code-Korrektheit (vgl. Kapitel 5) einhalten kann.

7.3.1 *Analyse von Interferenzen*

Dies wirft die Frage auf, woher diese Differenz zwischen der erzielten Korrektheit bei geringeren Schwellwerten bei Bottom Up und Sharpen und dem Höchstwert der Korrektheit bei $t_{ac} = 100\%$ bzw. $t_s = 0$ kommt. Eine stichprobenartige Sichtung der Ursachen bzw. der in diesen Fällen generierten Daten ergibt als primäre Begründung die Auswahl einer Hypothese, welche neben den jeweils dokumentierten Sperren noch weitere, irrelevante Sperren auswählt. Dieses vermutete Phänomen wird im Folgenden als *Interferenz* bezeichnet. Bei genauerer Untersuchung ergeben sich verschiedene Arten und mögliche Ursachen dieser Interferenzen.

7.3.1.1 *Arten und Ursachen von Interferenzen*

Zum einen ist es durch die allgemeine Funktionsweise der LocκDoc-Experimente zu erwarten, dass zu einem beliebigen Zeitpunkt eine gewisse Menge an Sperren gehalten werden, welche keine Relevanz für die zu untersuchenden Komponenten aufweisen. Diese lassen sich i. d. R. durch den hier geschehenden Einsatz von Kontext-sensitiver Analyse (vgl. Unterunterabschnitt 2.4.4.3) eliminieren.

Dem entgegen stehen Interferenzen innerhalb des aktuell analysierten Kontextes. Diese können u. a. dadurch auftreten, dass eine Routine mehrere Sperren zu Beginn ihrer Ausführung holt, welche zur korrekten Synchronisation aller stattfindenden Speicherzugriffe nötig sind und diese erst am Ende ihrer Laufzeit wieder freigibt. Es ist nicht trivial, diese Form von Interferenzen zu vermeiden.

Eine im Graphen ersichtliche Form zur Reduktion dieser Interferenzen besteht aus dem Anheben des Schwellwertes der Bottom Up-Strategie. Ersteres sorgt dafür, dass nur Sperren in der ausgewählten Hypothese berücksichtigt werden, welche in mindestens t_{ac} Prozent der Fälle auftreten. Somit können können diese Interferenzen zwar reduziert werden, ein zu weites Anheben dieses Schwellwertes (zur Vermeidung der Analyse von irrelevanten Sperren, welche sehr oft in den selben Routinen geholt werden), kann aber dazu führen, dass relevante Sperren, welche nicht fehlerfrei genutzt werden, nicht in die Analyse mit einfließen. Der bereits beschriebene starke Anstieg der Korrektheit der Bottom Up-Strategie bei hohem Schwellwert spricht allerdings dafür, dass dies kein großes Problem für NetBSD darstellt.

Eine analoge Schlussfolgerung lässt sich auf die Sharpen-Strategie anwenden.

Für LockSet ergibt sich dahingehend im Allgemeinen das gleiche Problem wie bei der Bottom Up-Strategie, da LockSet identisch zu Bottom Up mit $t_{ac} = 1$ als Schwellwert ist.

Die Top Down-Strategie ist von diesen Interferenzen weniger betroffen, da diese i. d. R. eine höhere Akzeptanzrate mit notfalls weniger Sperren präferiert. Diese Interferenzen werden im Bezug auf Top Down nur problematisch, wenn keine Zugriffe ohne diese irrelevanten geholten Sperren existieren.

Als alternative Methode zur Beseitigung einer Teilmenge dieser Interferenzen ist das getrennte Analysieren von unterschiedlichen Datenstrukturen nennenswert. Somit würden Fälle eliminiert werden, in denen bei Zugriffen auf eine Datenstruktur irrelevante Sperren einer anderen Datenstruktur geholt werden. Diese Taktik ist in der Praxis allerdings ungeeignet, da so keine Interferenzen *innerhalb* einer Datenstruktur und Interferenzen zu statischen Sperren vermieden werden können. Zudem würde diese Methode durch die wiederholten Experimente einen wesentlich höheren Zeitaufwand benötigen.

7.3.1.2 *Prüfen auf Existenz von Interferenzen*

Um zu Überprüfen, ob die aufgestellten Theorien über Interferenzen als Ursache für die suboptimale Korrektheit der Hypothesen wahr sind, wurden die Werkzeuge zur Kontrolle der Hypothesen modifiziert. Die bisher genutzte *strikte Prüfung* prüft, ob die ausgewählte Sperren-Regel die und nur die Sperren enthält, welche aus der Dokumentation übertragen wurden. Zusätzlich zu dieser Methode wurde im Rahmen dieser Arbeit die *Subset Prüfung* implementiert. Dieser Algorithmus untersucht die aufgestellten Hypothesen dahingehend, ob die dokumentierten Regeln eine Teilmenge dieser Hypothese bilden. Er eliminiert also den Fall, dass eine aufgestellte Hypothese aufgrund von konstant auftretenden Interferenzen als inkorrekt eingeordnet wird.

Zur besseren Veranschaulichung dieses Algorithmus folgt ein anwendungsnahes Beispiel. Es werden Schreibzugriffe auf das Mitglied `int vi_synclistslot` von `struct vnode_impl` betrachtet. Dessen Dokumentation (vgl. Listing 3) erklärt, dass vor jeglichen Zugriffen `kmutex_t syncer_data_slot` gesperrt sein muss. Es werden nun drei der aufgestellten Hypothesen untersucht:

1. `syncer_data_lock (100%)`: Diese Hypothese ist die laut Dokumentation korrekte und enthält keinerlei Interferenzen. Beide Methoden zum Prüfen auf Korrektheit von Hypothesen (strikt und Subset) akzeptieren diese.

2. `v_interlock -> syncer_data_lock (96.3%)`: Diese Hypothese enthält eine zusätzliche Sperre als Interferenz. Die strikte Prüfung würde diese Hypothese ablehnen, da sie nicht exakt die Menge an Sperren entspricht, welche dokumentiert ist. Die Subset-Prüfung hingegen akzeptiert diese Hypothese als korrekt, da sie alle dokumentierten Sperren enthält.

3. `v_interlock (96.3%)`: Diese Hypothese besteht ausschließlich aus Interferenz. Keine der Methoden zur Prüfung von Hypothesen akzeptiert diese als korrekt.

Eine Anwendung dieser zweiten Methode zum Prüfen auf Teilmengen von Sperren-Regeln ist ebenfalls in Abbildung 7.3.1 und Abbildung 7.3.2 zu sehen.

Aus diesen Resultaten, insb. aus der Differenz zwischen der strikten und der Subset-Prüfung, zeigt sich, welchen nennenswerten Einfluss solche Interferenzen auf die Auswahl einer korrekten Strategie zur Auswahl von Hypothesen haben. Ein überraschender Aspekt besteht auch daraus, dass diese Differenz selbst bei der Top Down-Strategie vorhanden ist. Daraus ergeben sich für einige Mitglieder der untersuchten Datenstrukturen zwei mögliche Erklärungen: Es existieren Interferenzen, welche in 100% der Zugriffe auf gewisse Datenstrukturen auftreten oder die dokumentierten Sperren-Regeln sind unzureichend.

Neben der Existenz von Interferenzen existieren noch weitere Ursachen für die Auswahl von inkorrekten Hypothesen. Diese werden in **??** genauer erläutert, aber auch Probleme bzgl. der Nutzung von Zeigern (vgl. Unterabschnitt 7.4.1), atomarer Zugriffe (vgl. Unterabschnitt 7.4.2) und Sperren-Regeln, welche durch LockDoc nicht abbildbar sind (vgl. Unterabschnitt 7.4.3), existieren.

Insgesamt zeigt sich aber, dass die Größenordnung der korrekten Sperren-Regeln vergleichbar mit denen aus vorangegangenen Arbeiten wie denen von Lochmann *et. al.* ([Loc21]) ist. Das lässt darauf schließen, dass auch wenn diese Ergebnisse nicht notwendigerweise optimal sind, die Implementierung keine Fehler gröberen enthält, welche etwaige Ergebnisse unbrauchbar machen würden.

7.3.2 *Auswahl einer Strategie*

Nachdem auf die Ursachen für die allgemein suboptimale Auswahl von Hypothesen eingegangen wurden, folgt nun die Auswahl einer Strategie für weitere stattfindende Analyse-Schritte.

Insgesamt ergibt sich prozentual gesehen für die Top Down-Auswahlstrategie der höchste Anteil an korrekt vorhergesagten Sperren-Regeln. In den folgenden Experimenten wurde daher diese Strategie verwendet. Lochmann *et. al.* nennen in [Loc21] anhand eines Beispiels den konzeptuellen Nachteil, der bei der Nutzung von Top Down Auftritt: Für Datenstrukturen und Sperren, welche ausschließlich gemeinsam gesperrt werden sollten, können Hypothesen ausgewählt werden, die zwar eine höhere Akzeptanzrate besitzen, allerdings aufgrund von Implementierungsfehlern nicht unbedingt korrekt sind. Dieses Verhalten konnte innerhalb der hier zu analysierenden Datenstrukturen nicht festgestellt werden. Dies ist insb. dadurch begründet, dass innerhalb der untersuchten Subsysteme nur eine sehr geringe Anzahl an Datenstrukturen existieren, bei denen mehr als eine Sperre zur korrekten Synchronisation benötigt wird.

Für sämtliche folgenden Schritte wurde aus den genannten Gründen die Top Down Strategie mit einem Schwellwert von $t_{ac} = 95\%$ gewählt. Dieser scheint einen geeigneten Mittelpunkt zwischen zu viel Interferenz und dem zu schnellen Verwerfen von inkorrekt genutzten Sperren zu bieten.

Ergebnisse der Auswahl dieser Strategie im Bezug auf die dokumentierten Sperren-Regeln sind in Tabelle 7.3.1 zu finden. Im Vergleich zu vorangegangenen Arbeiten werden in NetBSD wesentlich höhere Akzeptanzraten als in Linux erzielt, die gewonnenen Ergebnisse befinden sich etwa auf dem Niveau derer von FreeBSD. [Loc21, Tab. 7.10 u. 7.18]

7.4 BESONDERHEITEN

Bei der manuellen Auswertung der durch die Experimente erzielten Ergebnisse fallen einige Besonderheiten auf. Es wird auf eine Auswahl dieser genauer eingegangen.

7.4.1 Unklarheiten bzgl. Zeigern

Eine Komplikation, welche bei der Analyse auffällt, ist der Umgang mit Zeigern. Einerseits ist in der Dokumentation selten eindeutig, ob sich eine Sperre auf das Lesen/Schreiben des Zeigers bezieht oder auf den Zugriff auf den eigentlichen Inhalt der Datenstruktur, auf welche der Zeiger verweiset. Außerdem wird oftmals davon ausgegangen, dass Zeiger ohne jegliche Art von Sperren geschrieben werden können.

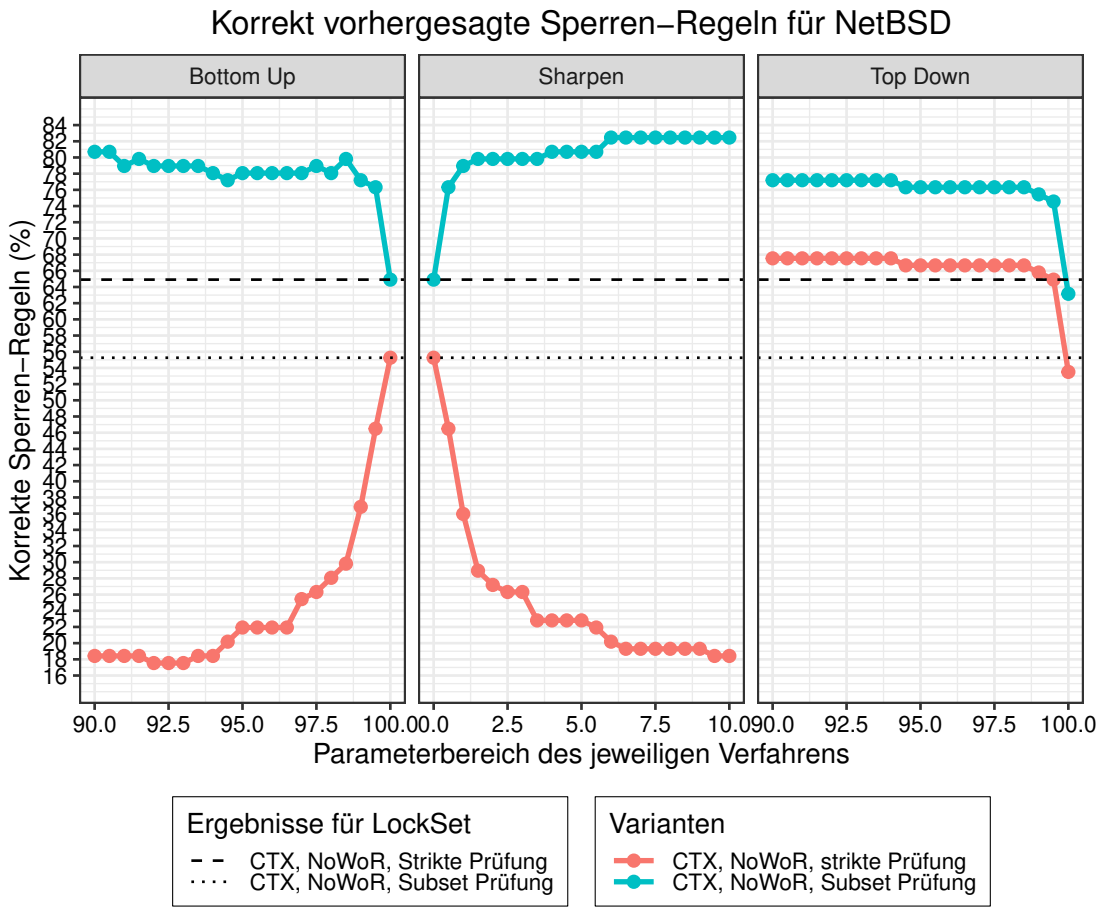

Abbildung 7.3.1: Vergleich der vier Auswahlstrategien zur Hypothesenbestimmung abhängig von ihren jeweiligen Parametern, angewandt auf den ATF-Datensatz.

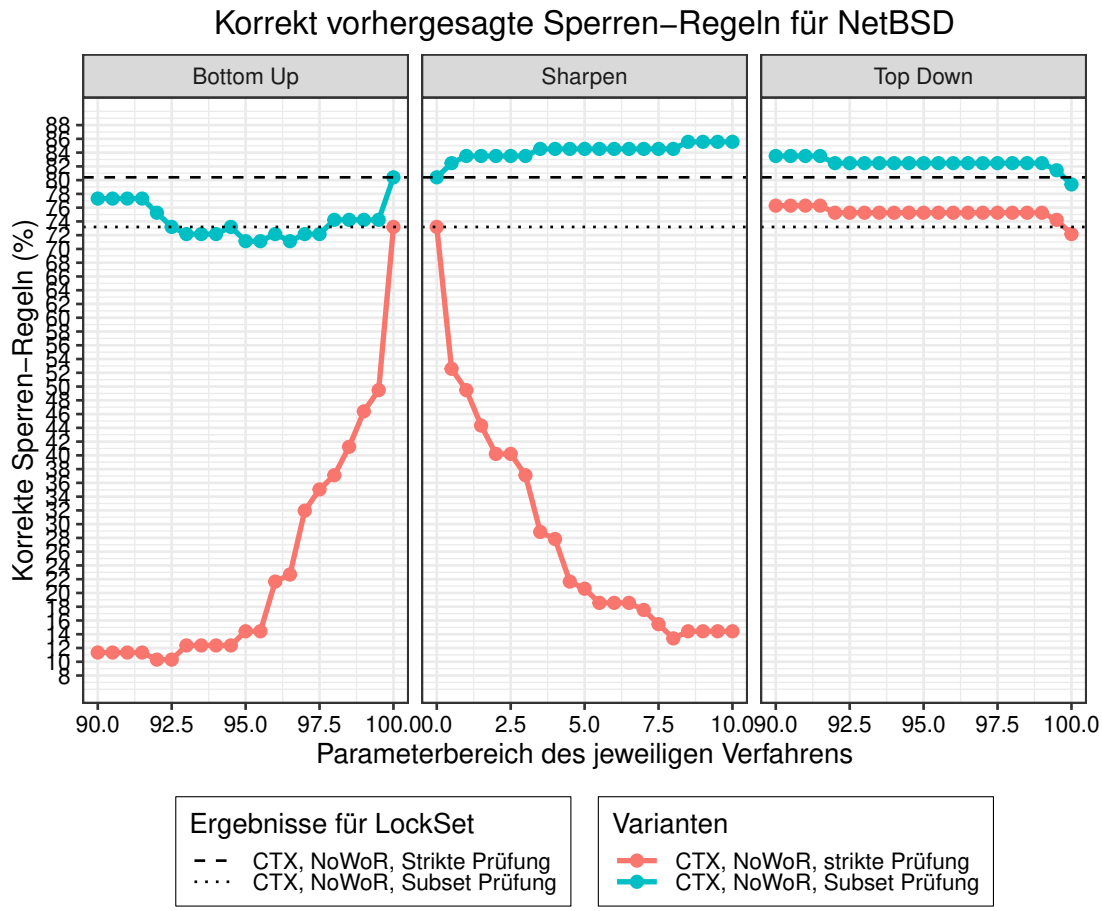

Abbildung 7.3.2: Vergleich der vier Auswahlstrategien zur Hypothesenbestimmung abhängig von ihren jeweiligen Parametern, angewandt auf den LTP-Datensatz.

Benchmark	Datentyp	#D	#Nb	#Be	✓ (%)	~ (%)	✗ (%)
ATF FS Test Suite	buf	56	2	54	68,52	31,48	0,00
ATF FS Test Suite	vnode_impl	80	20	60	63,33	28,33	8,33
LTP FS Test Suite	buf	56	8	48	87,50	12,50	0,00
LTP FS Test Suite	vnode_impl	80	31	49	73,47	16,33	10,20

Tabelle 7.3.1: Zusammenfassung der dokumentierten und bestätigten Sperren-Regeln. #D gibt die Anzahl der Regeln an, welche in eine Grundwahrheit überführt werden konnten. #Nb und #Be sind jeweils die absoluten Anzahlen dieser Regeln, welche nicht bestätigt bzw. bestätigt werden konnten. ✓, ~ und ✗ geben den prozentualen Anteil an Regeln an, welche jeweils immer gültig sind, nicht immer gültig sind und nie gültig sind.

7.4.2 *Unklarheiten bzgl. atomarer Zugriffe*

Ähnlich zu den Komplikationen bzgl. Zeigern existiert oft Unklarheit darüber, wie mit Integers (und damit auch Enums) umgegangen werden sollte. Diese besitzen zwar auch Dokumentation darüber, welche Locking-Vorkehrungen eingesetzt werden sollten, in der Praxis werden allerdings oft atomare Operationen zum Zugriff genutzt. Dieser Umstand ist bereits Lochmann *et. al.* aufgefallen:

> Im Austausch mit den Kern-Entwicklern wurde deutlich, dass Datentypen mit Wortgröße, wie z. B. int, aus Performanzgründen als atomar lesbar angenommen werden. [Loc21, Kap. 7]

Wird nun auf NetBSD-spezifische Fälle geblickt, so ist dies zwar selten im Bezug auf Integer der Fall, es kommt allerdings vermehrt vor, dass Zeiger und Enums ohne laut Dokumentation nötige Sperren gelesen oder gesetzt werden. Dies konnte in Absprache mit den Mitwirkenden bestätigt werden, stellt allerdings gewünschtes Verhalten dar.

7.4.3 *Freie Wahl von Lesersperren*

Eine weitere Komplikation ergibt sich durch eine spezielle Art von Sperren-Regel, welcher in vorherigen Arbeiten wenig Beachtung geschenkt wurde, allerdings vermehrt

in NetBSD auffällt: So existiert eine Anzahl an Sperren-Regeln, welche die freie Auswahl von Sperren für Lesezugriffe ermöglicht, zum Schreiben allerdings beide Sperren holt. Zwei Beispiele finden sich im `struct vnode` (vgl. Listing 4), dort heißt es:

> i+b v_interlock + bufcache_lock to modify, either to inspect
> i+u v_interlock + v_uobj.vmobjlock to modify, either to inspect

In den vom Hypothesizer aufgestellten Sperren-Regeln für Lesezugriffe ergeben sich somit Hypothesen mit einer geringen Supportrate für beide Sperren, diese werden dementsprechend nicht zu Gewinnerhypothesen erklärt. Stattdessen würde die Summe beider aufgestellter Regeln eine Supportrate von 100% besitzen, vorausgesetzt der Kernel beinhaltet keine fehlerhaften Zugriffe.

Diese Problematik wurde bereits in Abschnitt 3.5 beschrieben, diese Arbeit war jedoch nicht erfolgreich darin, dieses Problem zu beheben.

7.5 ERGEBNISSE

In Tabelle 7.5.1 ist ersichtlich, dass durch den Einsatz von ATF und LTP je 246.982 bzw. 307.837 Gegenbeispiele gefunden werden konnten, welche den durch den Hypothesizer ausgewählten Sperrenregeln widersprechen. Es fällt auf, dass durch LTP eine größere Anzahl dieser Gegenbeispiele ermittelt werden konnte. Dies kann möglicherweise durch die pure Anzahl an Stresstest innerhalb dessen Test Suite begründet sein. Die Anzahl der betroffenen Datenstruktur-Mitglieder ist bei ATF mit 30 jedoch größer als die von LTP erzielten 24. Eine mögliche Erklärung hierfür besteht darin, dass ATF dadurch, dass es nativ für NetBSD entwickelt wurde, dazu in der Lage ist, auf eine diversere Menge dieser Mitglieder zuzugreifen. Dies ist bspw. der Fall bei speziellen, NetBSD-spezifischen Dateisystemen, welche nicht von LTP unterstützt werden.

Durch die Analyse der ausgewählten Datentypen im VFS-Subsystem konnten bisher eine unklare Sperrendokumentation und ein Implementierungsfehler im NetBSD Kern identifiziert werden.

Benchmark	Datentyp	Häufigkeit	Mitglieder	Kontexte
ATF FS Test Suite	buf	110093	13	924
ATF FS Test Suite	mount	92425	6	81
ATF FS Test Suite	vnode_impl	44464	11	33
LTP FS Test Suite	buf	198374	7	454
LTP FS Test Suite	mount	22225	8	94
LTP FS Test Suite	vnode_impl	87238	9	225

Tabelle 7.5.1: Übersicht über die Anzahl an Gegenbeispielen (Speicherzugriffe, welche zu Verstößen der aufgestellten Sperrenregeln führen), die Anzahl der Datenstruktur-Mitglieder und die Anzahl der Kontexte. Alle Angaben je Benchmark und Datentyp.

7.5.1 *Unklare Dokumentation bei mehreren benötigten Sperren*

Die unklare Sperrendokumentation ist während des Vergleichens der existierenden (und in eine Grundwahrheit umgewandelten) Dokumentation mit den eigentlichen Sperrvorgängen aufgefallen. So liest sich in Listing 3[1]:

> n,l vi_nc_lock ① + vi_nc_listlock ② to modify

Dies kann auf unterschiedliche Weisen interpretiert werden, intuitiv ergeben sich zwei Möglichkeiten:

1. Die erste Variante zur Interpretation des Kommentars ist „(① + ②) to modify“. Hierbei ist keine Sperre zum Lesen nötig, beide Sperren zum Schreiben.
2. Als zweite Option besteht „① + (② to modify)“, es wäre also Sperre ① zum Lesen notwendig, beim Schreiben zusätzlich noch ②.

Das Übertragen keiner dieser Optionen in die Ground Truth führt zu einer hohen Akzeptanzrate für die relevanten Datenstrukturen.

[1] Code-Auszug befindet sich auf dem Stand vor Beseitigung des Problems

Ein Austausch mit NetBSD Mitwirkenden ergab anschließend, dass beide Interpretationen inkorrekt sind. Stattdessen sollte der Kommentar gelesen werden als: „Für einen Schreibzugriff sind beide Sperren nötig, für einen Lesezugriff lediglich eine beliebige dieser beiden Sperren“. Damit ergibt sich erneut das in Unterabschnitt 7.4.3 bereits bemerkte Muster.

Aufgrund dieser Verwirrung wurde der entsprechende Kommentar in CVS Revision 1.27[2] angepasst. Dieser liest nun:

> n,l both vi_nc_lock + vi_nc_listlock to modify, either to read

7.5.2 *Fehlende Sperre im FFS-Dateisystem*

Als Ergebnis der eigentlichen LockDoc-Analyse ergab sich bisher ein Fehler bzgl. des eigentlichen Lockings. Dieser befindet sich in der Implementierung des FFS-Dateisystems.

Die Durchsicht der gefundenen vermuteten Fehler in der Synchronisation zeigt als Hypothese für Lesezugriffe auf `vnode_impl.vi_vnode.v_numoutput` das Holen des `v_interlock`s. Diese Hypothese kann anhand der Dokumentation zur entsprechenden Datenstruktur bestätigt werden. Es existieren jedoch zwei Code-Pfade für welche dies nicht der Fall ist. Sowohl Aufrufe des Syscalls `sys_sync` als auch Aufrufe von `sys_unmount` rufen – sofern sie auf FFS-Dateisystemen ausgeführt werden – intern eine Methode zum syncen des Dateisystems auf, in diesem Fall `sys_unmount`. Keine der beiden Code-Pfade nutzt jedoch das `v_interlock`, was auf einen möglichen Bug hinweist.

Die Code-Stelle, welche auf das problematische Mitglied zugreift, findet sich in Listing 11.

In Zeile 4 wird überprüft, ob `vnode_impl.vi_vnode.v_numoutput` größer 0 ist. Dieser Lesezugriff dürfte nicht ungesichert stattfinden. Durch einem Austausch mit den NetBSD Mitwirkenden konnte dieser Bug bestätigt werden, er hat allerdings nur geringfügige Auswirkungen:

> <@Riastradh> although I guess it's not actually a big issue, because either
> <@Riastradh> (a) ffs_sync is being called concurrently with other file system

[2] http://cvsweb.netbsd.org/bsdweb.cgi/src/sys/sys/vnode_impl.h?rev=1.27&content-type=text/x-cvsweb-markup

```
/*
 * Force stale file system control information to be flushed.
 */
if (waitfor != MNT_LAZY && (ump->um_devvp->v_numoutput > 0 ||
    !LIST_EMPTY(&ump->um_devvp->v_dirtyblkhd))) {
        vn_lock(ump->um_devvp, LK_EXCLUSIVE | LK_RETRY);
        if ((error = VOP_FSYNC(ump->um_devvp, cred,
             (waitfor == MNT_WAIT ? FSYNC_WAIT : 0) | FSYNC_NOLOG,
            0, 0)) != 0)
                allerror = error;
        VOP_UNLOCK(ump->um_devvp);
}
```

Listing 11: Auszug aus `sys/ufs/ffs/ffs_vfsops.c`, CVS Revision 1.381

> activity, so new writes can be concurrently triggered anyway, so it doesn't really matter much; or
> <@Riastradh> (b) ffs_sync is being called when the file system is quiesced, in which case it can't change anyway.

Für den Bug wurde NetBSD Problem Report #57606[3] angelegt. Im Rahmen dieser Arbeit und in Absprache mit den Entwicklern wurde der Code so modifiziert, als dass der Fehler beseitigt wird. Dieser Vorgang beinhaltet das Aufteilen des if-Statements, da `v_interlock` nicht über `vn_lock` gehalten werden kann. Diese Lösung wurde in NetBSD als CVS Revision 1.382 akzeptiert. Der entsprechende Code ist in Listing 12 zu finden.

Die Tatsache, dass dieser Fehler in der Implementierung von FFS gefunden wurde, obwohl nur das VFS beobachtet wurde, bestätigt die in **??** aufgestellte Hypothese, dass durch das bloße Instrumentieren vom VFS in seiner Rolle als Interface zu sämtlichen Dateisystemen es möglich ist, Fehler in den Dateisystemen selbst zu finden.

Außerdem ist an dieser Stelle zu bemerken, dass das Auffinden dieses Fehler ohne die in Unterabschnitt 6.3.2 vorgestellten Verbesserungen von LockDoc nicht möglich gewesen wäre.

[3] https://gnats.netbsd.org/cgi-bin/query-pr-single.pl?number=57606

```
/*
 * Force stale file system control information to be flushed.
 */
if (waitfor != MNT_LAZY)  {
        bool need_devvp_fsync;

        mutex_enter(ump->um_devvp->v_interlock);
        need_devvp_fsync = (ump->um_devvp->v_numoutput > 0 ||
            !LIST_EMPTY(&ump->um_devvp->v_dirtyblkhd));
        mutex_exit(ump->um_devvp->v_interlock);
        if (need_devvp_fsync) {
                int flags = FSYNC_NOLOG;

                if (waitfor == MNT_WAIT)
                        flags |= FSYNC_WAIT;

                vn_lock(ump->um_devvp, LK_EXCLUSIVE | LK_RETRY);
                if ((error = VOP_FSYNC(ump->um_devvp, cred, flags, 0,
                            0)) != 0)
                        allerror = error;
                VOP_UNLOCK(ump->um_devvp);
        }
}
```

Listing 12: Auszug aus `sys/ufs/ffs/ffs_vfsops.c`, CVS Revision 1.382

8

FAZIT

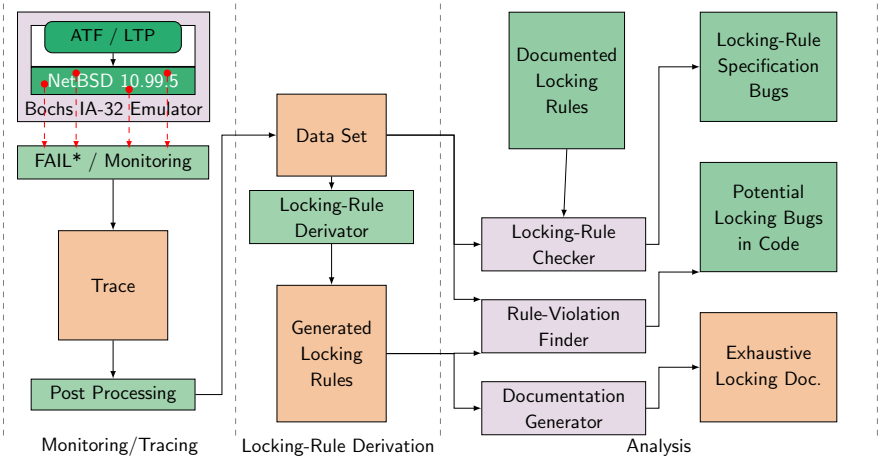

Abbildung 8.0.1: Beitrag dieser Arbeit zur LockDoc-Methode, aus [Loc21] adaptiert. Grün hinterlegte Felder wurden in dieser Arbeit portiert, angepasst, erweitert oder ausgewertet.

In dieser Arbeit wurde auf von Lochmann *et. al.* vorgestellte LockDoc-Methode zur automatisierten und experimentbasierten Sperrenanalyse so aufgebaut, dass diese auf den NetBSD-Kern angewandt werden kann. Durch diesen Einsatz von LockDoc auf ein Betriebssystem, welches hohen Wert auf Korrektheit und Dokumentation legt, konnten Schwachstellen von LockDoc identifiziert und zum Teil behoben werden.

Zu diesem Arbeitsaufwand gehören das Portieren und Integrieren der entsprechenden Kernel-Komponenten zu bzw. in NetBSD, gefolgt vom Instrumentieren von Datenstrukturen, welche im Folgenden genauer untersucht wurden.

Kernel-Modifikationen

Im Rahmen dieser Arbeit wurde der NetBSD-Kern in Version 10.99.5 dementsprechend modifiziert, dass jegliche Sperrenoperationen in Verbindung mit dem FAIL*-Emulator aufgezeichnet werden können. Des Weiteren ist es in diesem Kontext nun möglich, Speicherzugriffe auf beliebige Datenstrukturen zu beobachten. Damit ist der Grundbaustein für die eigentlichen LockDoc-Experimente gelegt worden. Außerdem erfolgen Modifikationen, um automatisiert das System und die entsprechenden Arbeitslasten starten zu können.

Anpassung und Erweiterung von LockDoc

Außerdem wurden die zu LockDoc gehörigen Werkzeuge so angepasst, als dass diese das neue Betriebssystem unterstützen.

Bei der Anpassung dieser Programme zur Datenverarbeitung fiel auf, dass eine in NetBSD sehr relevante Art von Sperren nicht von LockDoc unterstützt wird. Es folgte eine Evaluation über unterschiedliche Lösungsansätze und die Auswahl der geeignetsten Lösung. Dementsprechend wurden die entsprechenden Werkzeuge so erweitert, dass diese nun eine neue Art von Sperre verarbeiten können. Ohne diese Erweiterung war es nicht möglich, einen großen Teil der Mitglieder der Datenstrukturen in NetBSD zu untersuchen.

Instrumentierung

Anschließend erfolgte die Suche nach einem Kernel-Subsystem, welches sich für die Sperrenanalyse eignet. Analog zu vorangegangenen Arbeiten liefert das VFS-Subsystem einen idealen Startpunkt zur Evaluation. Dort wurden Datenstrukturen instrumentiert, welche anfällig für Race Conditions und Verklemmungen schienen: `vnode_impl`, `vnode`, `buf`, `mount` und in jene Strukturen verschachtelte bzw. direkt eingebettete Mitglieder.

Arbeitslast

In der Arbeit wurden zusätzlich verschiedene Arbeitslasten auf ihre Geeignetheit gegenüber diesem Anwendungszweck evaluiert. Die Auswahl fiel auf ATF zum allgemeinen Testen des Kernels und LTP zum Testen der Linux Emulationsschicht und zur besseren Vergleichbarkeit mit den vorangegangenen Arbeiten.

Evaluation

Anschließend folgte die begründete Auswahl der Parameter wie bspw. der Algorithmus zur Hypothesengenerierung, welche zur Nachverarbeitung genutzt wurden.

Im Zuge dessen ist eine weitere Schwachstelle von LockDoc aufgefallen: Die Behandlung von auftretenden Interferenzen. Die vorhandenen Werkzeuge wurden erweitert, um diese Problematik genauer untersuchen, und somit dessen Ausmaß feststellen zu können. Als letzte Problematik von LockDoc ist der Umgang mit non-trivialen Sperren-Regeln nennenswert.

Ergebnisse

Anhand der so gewonnenen Ergebnisse konnten Unklarheiten in der NetBSD-Dokumentation und Fehler in der Implementierung innerhalb des VFS-Subsystems gefunden und in Kooperation mit den NetBSD-Mitwirkenden behoben werden.

In Abbildung 8.0.1 ist eine Übersicht über die modifizierten und erweiterten Komponenten anhand des Flussdiagramms aus Abbildung 2.4.1 dargestellt.

8.1 FAZIT

Die vorgestellte Arbeit hat mehrere Ergebnisse erzielen können:

Zum einen wurde aufgezeigt, wie LockDoc auf neue, bisher nicht unterstütze Betriebssysteme portiert werden kann. Im Zuge dessen wurde analysiert und festgestellt, dass dies einen vergleichsweise geringen Arbeitsaufwand mit sich bringt. Dadurch sollte es die Hemmschwelle für zukünftige Arbeiten senken, die LockDoc Methode auf wieder neuen Betriebssystemen zu implementieren.

Weiterhin konnten die mit LockDoc verbundenen Werkzeuge dahingehend verbessert werden, dass diese eine bisher nicht unterstütze Art von Sperre in der Nachverarbeitung miteinbeziehen können, ohne diese explizit instrumentieren zu müssen. Es wurden zudem weitere Schwachstellen von LockDoc genauer untersucht.

Durch diese erfolgreiche Erweiterung war es möglich, Fehler und Dokumentationsprobleme in der Implementierung in NetBSDs VFS-Subsystem automatisiert zu finden und zu beheben. Ohne die in dieser Arbeit zu LockDoc hinzugefügte Funktionalität wäre dies nicht möglich gewesen. Im Allgemeinen konnte zudem bestätigt werden,

dass NetBSD, welches laut eigenen Angaben viel Wert auf Korrektheit legt, dieses Versprechen auch weitestgehend einhalten kann.

Zusammenfassend konnten NetBSD und LockDoc dahingehend kombiniert und erweitert werden, um das jeweils andere Projekt zu verbessern.

8.2 ZUKÜNFTIGE ARBEITEN

Im Laufe dieser Arbeit sind einige Szenarien aufgefallen, in denen der LockDoc Ansatz nicht ohne Probleme eingesetzt werden konnte. Während einige dieser Probleme gelöst werden konnten, so existiert noch keine Lösung für das Behandeln von Sperren-Regeln, welche die freie Auswahl zwischen mehreren Locks überlassen (vgl. Unterabschnitt 7.4.3). Die Unterstützung von Synchronisation über Referenzzählung stellt eine weitere, noch nicht vorhandene Funktionalität dar. Außerdem bleibt die Behandlung von auftretenden Interferenzen (vgl. Unterabschnitt 7.3.1) eine allgemeine Schwachstelle des Ansatzes.

In dieser Arbeit wurde zwar auf die Codeabdeckung der vorherigen LTP-basierten Arbeitslasten eingegangen, die Analyse über die in NetBSD (insb. in Verbindung mit ATF) erreichte Basisblock-Abdeckung verbleibt allerdings auch für zukünftige Arbeiten.

Im Kontext dieser genaueren Analyse der Arbeitslast steht die Evaluation von anderen möglichen Arbeitslasten und Methoden zum automatisierten Erzeugen dieser (bspw. durch Fuzzing, vgl. Abschnitt 4.1) weiterhin aus.

Auch die Analyse von unerforschten Subsystemen stellt ein mögliches Forschungsziel da, nachdem ein Großteil des Aufwandes der Portierung von LockDoc auf NetBSD in dieser Arbeit absolviert wurden. Es wurde aber auch ein Überblick über diesen benötigten Aufwand geschaffen, was die Portierung und den Einsatz von LockDoc auf weiteren, nicht erforschten Betriebssystemen vereinfacht.

LITERATURVERZEICHNIS

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[Chu16] SungWon Chung. The design of the netbsd I/O subsystems. *CoRR*, abs/1605.05810, 2016.

[Cor11] Jonathan Corbet. The real bkl end game, Jan 2011.

[Dev10] The NetBSD Developers. *ATF(7), NetBSD Manual Pages*. The NetBSD Foundation, Aug 2010.

[Dev11] The NetBSD Developers. Netbsd internals, Apr 2011.

[Dev16a] The NetBSD Developers. *pserialize(9), NetBSD Manual Pages*. The NetBSD Foundation, Jan 2016.

[Dev16b] The NetBSD Developers. *psref(9), NetBSD Manual Pages*. The NetBSD Foundation, Apr 2016.

[Dev17] The NetBSD Developers. *mutex(9), NetBSD Manual Pages*. The NetBSD Foundation, Dec 2017.

[Dev20] The NetBSD Developers. *condvar(9), NetBSD Manual Pages*. The NetBSD Foundation, May 2020.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, oct 2003.

[EM04] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 191–210, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 219–232, New York, NY, USA, 2000. Association for Computing Machinery.

[fre] Freebsd manual pages - smp.

[GCP99] Orna Grumberg, EM Clarke, and D Peled. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science; Springer: Berlin/Heidelberg, Germany*, 1999.

[HM94] David Helmbold and Charles Edward McDowell. *A taxonomy of race detection algorithms*. Citeseer, 1994.

[Loc21] Alexander Lochmann. *Aufzeichnungsbasierte Analyse von Sperren in Betriebssystemen*. PhD thesis, 2021.

[LS21] Alexander Lochmann and Horst Schirmeier. Beastie in for checkup: Analyzing freebsd with lockdoc. In *Tagungsband des FG-BS Herbsttreffens 2021*, Bonn, 2021. Gesellschaft für Informatik e.V.

[LSBS19] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[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.

[SB+15] Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, nov 1997.

[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 *2015 11th European Dependable Computing Conference (EDCC)*, pages 245–255, 2015.

[Sta17] William Stallings. *Operating Systems Internals and Design Principles, Global Edition*. Pearson Deutschland, 2017.

[Wat07] Robert N. M. Watson. Before & after under the giant lock, Nov 2007.