# Fehlertolerante Systemaufrufe mit einem Linux-Kern-Modul

Luca Kleinschmidt

Forschungsseminar: Trends in Betriebssystemen - Wintersemester 2024/25
Betreut von Lukas Pirl
Professur für Betriebssysteme und Middleware
luca.kleinschmidt@student.hpi.uni-potsdam.de

Fehlschlagende, aber nicht korrekt behandelte Systemaufrufe können zu Ausfällen von Anwendungen führen. Transiente Fehler können durch einen Wiederversuch des Systemaufrufs vor der Propagierung des Fehlzustands an die Anwendung toleriert werden. Das in dieser Arbeit entwickelte Kern-Modul für *Linux* 4 kann die Wiederversuchslogik in die Systemaufruftabelle laden. Die Wiederversuchslogik ist auf spezifische Fehlermodelle und damit Anwendungsfälle anpassbar und lässt sich perspektivisch durch beschriebene Heuristiken optimieren. Eine Beispielanwendung, welche Wettlaufbedingungen in der Speicherreservierung provoziert, demonstriert die Funktionsfähigkeit des Systems. Messungen zeigen einen geringen Mehraufwand von ungefähr 1 % der Wiederversuchslogik bei Prüfung auf einen Fehlerzustand.

## 1 Einführung

Die initiale Idee des Betreuers basiert auf dem Paper „Dependability Assessment of the Android OS Through Fault Injection“ [2], welches für das Android-Betriebssystem untersucht hat, wie von dem Betriebssystem und der Middleware propagierte Fehler durch Ausnahmebehandlung in verschiedenen Anwendungen behandelt werden. Einige der injizierten Fehler wurden in verschiedenen Anwendungen nicht behandelt und führten zu einem Ausfall der Anwendung. Eine Übertragung in den Kontext der verlässlichen Ausführung unter *Linux* führte zu der Idee, fehlgeschlagene Systemaufrufe innerhalb eines Kern-Moduls erneut auszuführen, um eine Behandlung von transienten Fehlern für alle Systemaufrufe eines Typs umzusetzen. Die Erkennung von Fehlzuständen erfolgt durch Prüfung und Behandlung von Ausnahmen und Rückgabewerten der Schnittstellenbeschreibung. Das Projekt *Kernel Randomized Faulter* überschreibt die Systemaufruftabelle mit Zeigern auf eigene Funktionen, welche wiederum Fehler injizieren [4]. Die Verwendung dieses Musters wurde genutzt, um eine Wiederversuchslogik analog zu einer Injektionslogik für *Linux 4.X* als Kern-Modul zu implementieren. Sicherheitsmechanismen verhindern eine Verwendung mit neueren *Linux*-Versionen. Basierend auf einer zeitbasierten Wiederversuchslogik wird eine Heuristik vorgeschlagen, welche nur bei konkreten aussichtsreichen Systemaufrufrückgabewerten einen Wiederversuch vornimmt. Bei der Vermessung konnte sowohl die Effektivität der Wiederversuchslogik als auch der geringe Mehraufwand gezeigt werden. Auch

besprochen werden Probleme wie Seiteneffekte oder fallspezifische Behandlungen von Rückgabewerten.

## 2 Verwandte Arbeiten

Diese Arbeit verwendet gleiche Mechanismen wie das Projekt *Kernel Randomized Faulter*, um Systemaufrufe abzufangen. *Kernel Randomized Faulter* agiert hingegen als Fehlerursache (nicht als Toleranzmechanismus) und injiziert Fehler, um beispielsweise Anwendungen auf korrekten Umgang mit fehlerhaften Standardbibliotheksaufrufen zu testen [4].

„FIRestarter: Practical Software Crash Recovery with Targeted Library-level Fault Injection" verwendet Sicherungspunkte, um bei einem Fehlerzustand zu einem Punkt zurückkehren zu können, der den Fehlzustand mutmaßlich noch nicht beinhaltet. Dafür wird jedoch eine Anpassung des Compilers notwendig, um spezielle Instruktionen einzubauen, welche für das Wiederherstellen verwendet werden. Transiente Fehlerursachen können durch ein Wiederherstellen und erneutes Ausführen behoben werden. Permanente Fehler werden durch die Propagierung einer Ausnahme an die Anwendung gemeldet, welche die Anwendung ihrerseits behandeln muss, um einem Ausfall vorzubeugen. Um diese Mechanismen einzubauen, wird ein Proxy für die Standardbibliotheken vorausgesetzt. Die Idee hinter diesem Konzept ist, dass die Kodierung zur Ausnahmebehandlung nicht den gleichen Kontrollfluss nutzt und somit die Fehlerursache des harten Fehlers nicht verwendet. Hierfür ist die Annahme notwendig, dass die Anwendungssoftware korrekt geschrieben ist und alle möglichen Ausnahmen behandeln kann. Ähnlich zu dem Vorgehen von „FIRestarter: Practical Software Crash Recovery with Targeted Library-level Fault Injection" sind „Rx: treating bugs as allergies—a safe method to survive software failures"[9] sowie „ASSURE: automatic software self-healing using rescue points"[10] und „REASSURE: A Self-contained Mechanism for Healing Software Using Rescue Points"[8], weswegen auf diese nicht weiter eingegangen wird.

Im Bereich des wissenschaftlichen Rechnens können Berechnungen über lange Zeit laufen und teure Ressourcen binden. Daher beschäftigt sich „A Case for Virtual Machine Based Fault Injection in a High-Performance Computing Environment" mit dem Interesse an verlässlicher Ausführung in Umgebungen des wissenschaftlichen Rechnens. Da diese Umgebungen normalerweise nicht vom Anwender administriert und mit bestimmten Programmversionen bereitgestellt werden, muss genau diese Kombination auch verwendet werden. Somit schließen alle Fehlertoleranzmechanismen aus, welche erhöhte Berechtigungen voraussetzen. Über die Umgebungsvariable *LD_PRELOAD* können dynamische Bibliotheken injiziert und damit Systembibliotheken überschrieben werden [5].

Alle betrachteten Ansätze „Dependability Assessment of the Android OS Through Fault Injection", „FIRestarter: Practical Software Crash Recovery with Targeted Library-level Fault Injection" und „A Case for Virtual Machine Based Fault Injection in a High-Performance Computing Environment" fangen die Propagie-

rung des Fehlerzustandes im Nutzermodus ab und behandeln ihn dort. Diese Arbeit wird auf die Behandlung des Fehlzustandes analog zu *Kernel Randomized Faulter* eine Behandlung im Kern vornehmen.

# 3 Fehlermodell

Die Fehlerklassen können lediglich auf die möglichen Rückgabewerte von Systemaufrufen begrenzt werden. Dabei sind Rückgabewerte notwendiges, aber nicht hinreichendes Kriterium für die Unterscheidung zwischen harten oder transienten Fehlern. Wenn beispielsweise ein Gerät als beschäftigt angegeben wird, kann das Gerät kurze Zeit später wieder verfügbar sein. Anders verhält es sich hingegen bei *Gerät nicht erreichbar*: Das Gerät kann entweder nur kurzzeitig nicht verbunden oder dauerhaft defekt sein. Das Wiederversuchen und damit Tolerieren eines Fehlers soll ausdrücklich nicht das Wählen eines anderen Pfades im Informationsfluss der Anwendung provozieren. Ziel eines Wiederversuchs ist das Behandeln von transienten Fehlern bei Systemaufrufen.

## 3.1 Auswahl der Systemaufrufe für ein angepasstes Fehlermodell

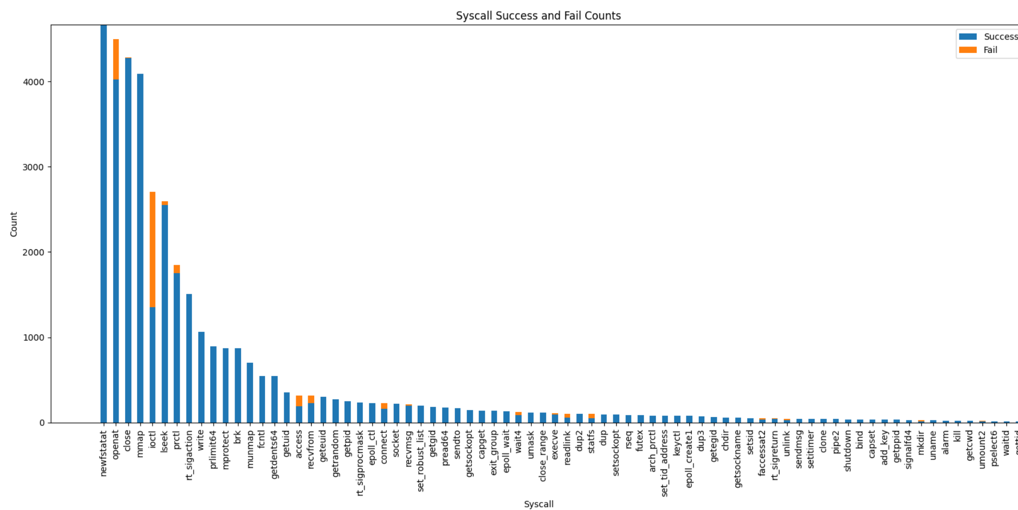


**Abbildung 1:** Auszug aus einer Systemaufrufübersicht eines Servers erstellt mit *auditctl*. Der Systemaufruf *ioctl* schlägt häufig ohne Fehlzustände fehl - das Hinzufügen eines Proxys mit Wiederversuchslogik kann die Responsivität des Systems daher negativ beeinflussen.

Da verschiedene Systemaufbauten verschiedene Ausprägungen der Fehlerklassen besitzen, empfiehlt sich ein Anpassen an die konkreten Umstände und relevanten Systemaufrufe. *Relevanz* kann hierbei nicht generisch definiert werden, da dies individuell und systemabhängig definiert werden muss. Anhaltspunkte für eine Anpassung des Fehlermodells und der Fehlerklassen können hohe oder geringe Diversität in Rückgabewerten sowie ein häufiges oder seltenes Fehlschlagen der Systemaufrufe liefern. Selten bis nie fehlschlagende Systemaufrufe können für Randfälle relevant sein und sind daher auch eine Betrachtung wert.

Mit dem Befehl

```
sudo auditctl -a always,exit -S all
```

kann die Aufzeichnung der Systemaufrufnutzung gestartet werden. Die Berichte über erfolgreiche und fehlgeschlagene Systemaufrufe können dann mit

```
sudo aureport --syscall --summary --interpret --failed
sudo aureport --syscall --summary --interpret --success
```

generiert werden. Diese Berichte können mit einem *Python*-Programm in eine Grafik umgewandelt werden (Abbildung 1).

## 3.2 Heuristik zur Optimierung

| Error / Syscall | EINVAL | ENOSPC | EINTR | ENOMEM | ENOTTY | ENOENT |
|---|---|---|---|---|---|---|
| write | n | j | j | - | - | - |
| read | n | - | j | - | - | - |
| mmap | n | - | j | j | - | - |
| ioctl | n | - | - | - | n | - |
| open | n | j | j | j | - | j |

**Tabelle 1:** Auszug aus einer beispielhaften Heuristik. Die Fehlerwert repräsentieren Fehlerklassen und verweisen somit auf mögliche Fehler(klassen). Die Tabelle beschreibt die Logik der Heuristik zur Fehlervermeidung. [j]a/[n]ein gibt an, ob ein Wiederversuch den Fehler beheben könnte und deswegen sinnvoll ist. Ein Strich [-] bedeutet, dass der Fehlerwert für den Systemaufruf nicht relevant ist also in der Dokumentation für den Systemaufruf nicht vorkommen. Siehe: errno.h,[1] errno-base.h[2] und *man* für den jeweiligen Systemaufruf.

Durch eine Heuristik verändert sich der Ansatz des Fehlermodells. Bisher wurde bei jedem Fehlschlag ein Wiederversuch vorgenommen. Wenn nun der Rückgabewert *ENOTTY*, also kein Ausgabegerät, fehlschlägt, ist ein Wiederversuch nicht sinnvoll, da das verwendete Gerät nach 10 Millisekunden immer noch kein Ausgabegerät ist. Der Rückgabewert *ENOMEM*, nicht ausreichend Arbeitsspeicher vorhanden, kann bei einem Wiederversuch Erfolg haben, siehe Abschnitt 5. Eine

| Syscall \ Error | EPIPE | EINVAL | EINTR | ENOSPC |
|---|---|---|---|---|
| write | y | n | y | y |

**Tabelle 2:** Analog zu Tabelle 1 eine Heuristik für eine Anwendung mit Problemen beim Schreiben auf einem Gerät.

Heuristik kann für jeden *relevanten* Systemaufruf und die, für diesen, einschlägigen Rückgabewerte erstellt werden. Ein Ansatz für solch eine Heuristik ist in Tabelle 1 gegeben.

Exemplarisch ist eine Anwendung gegeben, die über Dateizeiger mit einem Gerät kommuniziert, welches Probleme mit dem Lesen auf dem Bus besitzt. Tabelle 2 zeigt daher eine Heuristik für *write*, welche spezifisch auf Fehler des Busses angepasst ist. Die Heuristik kann durch das Einsparen von unnötigen Wiederversuchen die Geschwindigkeit des Systems verbessern. Jedoch ist mit der Heuristik weiterhin keine Unterscheidung zwischen harten und transienten Fehlern möglich, lediglich bei den durch die Heuristik behandelten Fehlerwerten wo ein *kein Wiederversuch* eingetragen wird, ist diese Klassifizierung fallbasiert möglich, jedoch nicht notwendig.

# 4 Kern-Modul

Für *Linux* 4.X wurde ein Kern-Modul entwickelt, welches Wiederversuchslogik in die Systemaufruftabelle des Kerns während des Betriebs einsetzen kann. Dabei wird der Zeiger einer Funktion mit Wiederversuchslogik an die Stelle des originalen Systemaufrufzeigers gesetzt. Der ursprüngliche Zeiger wird zur späteren Rekonstruktion beim Entladen des Kern-Moduls und für das Aufrufen des eigentlichen Systemaufrufs gespeichert.

Das initiale Beispiel ist minimal gehalten. In der Initialisierungsfunktion des Moduls wird mit *kallsyms_lookup_name* (Unterunterabschnitt 4.2.2) der Zeiger auf die Systemaufruftabelle abgerufen. Dann wird mit *write_cr0* der Schreibschutz auf den Speicherbereich der Systemaufruftabelle aufgehoben und mittels Zeigerarithmetik der ursprüngliche Zeiger in einen temporären Zwischenspeicher geschrieben, um ihn später rekonstruieren zu können. Dann wird der Zeiger durch einen Zeiger einer Proxyfunktion ersetzt. Die Wiederversuchslogik in der Proxyfunktion greift wiederum auf den originalen Zeiger im Zwischenspeicher zu und führt den Systemaufruf dort aus. Wenn dieser mit einem Fehlerwert beendet wird, wird beispielsweise nach 10, 50 und 100 Millisekunden ein erneuter Versuch gestartet. Wenn diese auch fehlschlagen, gilt der Systemaufruf als endgültig fehlgeschlagen und der Fehlerzustand wird an den Aufrufer weitergegeben. Eine Heuristik zur Optimierung des Vorgehens wird in Unterabschnitt 3.2 beschrieben. Wenn ein Versuch erfolgreich ist, wird der Rückgabewert direkt an den Aufrufer weitergegeben. Die Parameter werden über die Struktur *pt_regs* übergeben, wo der Zeiger direkt

an den Proxy übergeben und von diesem direkt an den eigentlichen Systemaufruf weitergegeben werden kann. Hier muss auf einer Fallbasis auf Seiteneffekte geachtet werden, wie bei *read*, wo in das per Referenz übergebene Argument *buf* Daten geschrieben werden.

Beim Entladen wird der modifizierte Zeiger gegen den ursprünglichen Zeiger getauscht und das Kern-Modul wird beendet. Für die Wettlaufbedingung bei der Entladung siehe Unterabschnitt 4.1.

Somit ist ein Wrappen eines generischen Systemaufrufs nun möglich. Um nun mit beliebig vielen Systemaufrufen kompatibel zu sein, wurde versucht, mit einem Präprozessormakro Quellen zu generieren. Nach dem Scheitern dieses Versuches wurde ein *Python*-Programm geschrieben, welches die geforderten, repetitiven C-Quellen für eine Liste von Systemaufrufnummern generiert.

## 4.1 Probleme beim Entladen des Kern-Moduls

Beim Entladen des Kern-Moduls sind vermehrt Speicherzugriffsfehler aufgetreten, die zu einer Kernpanik geführt haben. Eine mögliche Erklärung dafür ist eine Wettlaufbedingung bei Systemaufrufen, wie im Folgenden beispielhaft skizziert:

1. Anwendung führt Systemaufruf aus.
2. Kern lädt Funktionszeiger des Kern-Moduls aus der Systemaufruftabelle
3. Entladen des Moduls mit *rmmod* angefordert. Systemaufruftabelle wird mit originalen Zeigern überschrieben, Funktionen mit Wiederversuchslogik aus dem Speicherraum des Kerns entfernt.
4. Funktion an geladenem Zeiger wird ausgeführt. Da die Funktion aus dem Kernspeicherbereich entladen wurde, schlägt der Aufruf fehl.

Dies kann einfach mit einer Wartezeit zwischen dem Entladen des Moduls und dem Schließen des Kern-Moduls erfolgen. Die Wartezeit muss hoch genug gewählt werden, um sicherzustellen, dass das Kern-Modul beendet wird, nachdem alle Systemaufrufe beendet sind, die vorgeladene Proxy verwenden.

## 4.2 Sicherheitsmechanismen der Systemaufruftabelle in Linux 5.X und neuer

In der Linuxentwicklung haben verschiedene Sicherheitsmechanismen Einzug gehalten, die eine Verwendung des Kern-Moduls unter neuen *Linux*-Versionen nicht möglich machen. Einige können einfach umgangen werden, andere sind schwer bis unmöglich zu umgehen. Für das Verwenden eines Proxys ist dies problematisch. Bei einem direkten Einbau der Wiederversuchsfunktionalität direkt in den Kern sind diese Hindernisse nichtig.

### 4.2.1 Schreibschutz

Als Folge verschiedener Sicherheitslücken, unter anderem *CVE-2017-7308*, wurde das *cr4*-Register schreibgeschützt [12]. Da analog zu *cr4* auch mit dem *cr0*-Register Angriffe möglich sind, wurde dieses ebenfalls geschützt [11].

Für unter anderem Kern-Module (Ring 0) kann jedoch nach kurzer Internetrecherche eine Maschienenanweisung gefunden werden, die das sechzehnte *cr0*-Bit setzen. Somit den Schreibschutz für schreibgeschützte Seitentabellen [3], welche die Systemaufruftabelle beinhalten, aufheben kann.[3] Somit ist ab *Linux* 5.3 eine Umgehung notwendig.

### 4.2.2 Symbole nicht Sichtbar

Die Funktion *kallsyms_lookup_name* ermöglicht ein Finden des Systemaufruftabellenzeigers. Da diese Funktion jedoch nicht in Kern-Modulen innerhalb des Quellbaums des Kerns verwendet wird und potenziellen Angreifern eine Erleichterung bietet, wird das Symbol nicht mehr innerhalb des Kerns für Kern-Module exportiert [7]. Diese Änderung ist ab Kern 5.7 wirksam.

Auch hier gibt es wieder eine Möglichkeit, den Zeiger auf die Funktion *kallsyms_lookup_name* ohne Zuhilfenahme des Symbols zu erhalten.[4]

### 4.2.3 Umbau der Abrufmethode der Systemaufrufzeiger

Der Schreibschutz auf den Seitentabellen Unterunterabschnitt 4.2.1 sowie das Abrufen der *sys_call_table*-Symbole Unterunterabschnitt 4.2.2 konnte umgangen werden. Jedoch wurde aufgrund potenzieller Angriffe auf die Sprungvorhersage die Systemaufruftabelle, welche bisher über Zeigerarithmetik erfolgte, in eine mit C-Makros erstellte *switch*-Logik überführt [6]. Ein Überschreiben der Tabelle ist weiterhin möglich, jedoch unwirksam. Somit ist zwar theoretisch ein Ausführen des falschen Systemaufrufs möglich, jedoch können keine zufälligen Speicherbereiche angesprungen und ausgeführt werden. Zum einfachen Erstellen von Prototypen wurde daher *Debian* 10 mit *Linux* 4.X verwendet, da dort ohne komplexe Umgehungsstrategien getestet werden konnte.

Da diese Arbeit das Wiederversuchen von Systemaufrufen untersucht und explizit nicht das Finden von Umgehungen für Sicherheitsstrategien wurden Konzepte zum Überschreiben der Speicherbereiche der *switch*-Anweisung verworfen. Ein Überschreiben der einschlägigen Speicherbereiche ist zudem nur schwer generisch möglich, da, wie die Beschreibung des Beitrags in der *Linux*-Versionsverwaltung ausführt, die Komplexität von verschiedenen Optimierungen der Übersetzungseinheit abhängt. Für den Produktionseinsatz ist daher nur ein Direkteinbau in den Kern möglich und ein dynamisches Laden oder Entladen nur mit einem in den Kern eingebauten Haken möglich.

---

[3] `https://jm33.me/we-can-no-longer-easily-disable-cr0-wp-write-protection.html`
[4] `https://github.com/xcellerator/linux_kernel_hacking/issues/3`

#### 4.2.4 Verwendung in einer Produktionsumgebung

Wie in den vorangegangenen Sektionen beschrieben, ist ein dynamisches Laden und Entladen in aktuellen *Linux*-Versionen nicht mehr möglich. Für eine Umsetzung muss daher der Kern modifiziert werden. Die einfachste Lösung sieht einen direkten Einbau in die Logik des Kerns vor. Jedoch kann der Kern auch modifiziert werden, um einen Haken für Kern-Module anzubieten und ein Laden und Entladen zu ermöglichen. Dies würde jedoch die in Unterunterabschnitt 4.2.2 und Unterunterabschnitt 4.2.1 beschriebenen Sicherheitslücken erneut öffnen.

Dort wird dann analog zu Unterabschnitt 3.2 eine Heuristik mit Wiederversuchslogik für den Einzelfall eingebaut.

## 5 Vermessung

Zur Demonstration der Funktionsfähigkeit der Wiederversuchslogik wurde eine Beispielanwendung entwickelt. Alle Messungen werden auf einer Debian 10 *QEMU* virtuellen Maschine mit *Linux* 4.19.0-27-amd64 durchgeführt. Es stehen dem System 256 Mibibyte Arbeitsspeicher und ein CPU-Kern eines AMD Ryzen 5 5600G zur Verfügung.

Der Konfigurationsparameter *vm.overcommit_memory* wurde deaktiviert, um das System zur Speicherreservierung bei *malloc* zu zwingen, sodass der Speicher nicht erst beim ersten Speicherzugriff reserviert wird. Dies stellt sicher, dass der Systemaufruf *mmap* fehlschlägt, wenn nicht ausreichend Arbeitsspeicher verfügbar ist. Die Testanwendung erzeugt eine Wettlaufbedingung, die durch erneutes Versuchen des Systemaufrufs behoben werden kann. Die Wettlaufbedingung entsteht, wenn zwei verschiedene Threads gleichzeitig versuchen 128 Mibibyte Speicher zu beanspruchen. Basierend auf der Systemzeit wird zu einem geraden Takt von einem Thread Speicher freigegeben und gleichzeitig von einem anderen Thread reserviert. Zu einem ungeraden Takt verhält sich dies genau umgekehrt. Somit kann es passieren, dass der Arbeitsspeicher zuerst angefragt wird bevor er freigegeben wird. Somit schlägt in diesen Fällen *malloc* fehl, da die virtuelle Maschine maximal einmal 128 Mibibyte auf dem Stapel zur Verfügung stellen kann, da das System Teile des Arbeitsspeichers benötigt. Ohne Wiederversuchslogik scheiterte der Prozess beim zweiten Takt. Die Wiederversuchslogik ist in der Lage die regelmäßig auftretenden Wettlaufbedingungen der beiden Threads zu lösen.

Die Strategie für die Vermessung des Mehraufwandes des Fehlertoleranzmechanismus beinhaltet lediglich die Messung des Mehraufwandes im Nicht-Fehler-Fall. Der Fall, in dem ein Fehlerzustand auftritt und behandelt wird, wird als für die Vermessung nicht relevant betrachtet, da der Mehraufwand durch zusätzliche Instruktionen im Vergleich zu den Wartezeiten vor dem Wiederversuch zu vernachlässigen ist. Die Wartezeit ist im Vorhinein durch das Fehlermodell bekannt und daher kalkulierbar. Ein C-Programm, welches 8192 mal 128 Mibibyte Arbeitsspeicher reserviert und anschließend freigibt, misst pro Iteration die Dauer der Speicherreservierung in CPU-Zyklen und berechnet den Durchschnitt über alle

Speicherreservierungen hinweg. Dieses Programm wird 512-mal ausgeführt und ein Durchschnitt über alle Durchschnitte gebildet.

Gemessen wurde einmal ohne und einmal mit Wiederversuchslogik für den Systemaufruf *mmap*. Als Basiswert wurden im gerundeten Durchschnitt 147 652 Zyklen gemessen. Mit einer Wiederversuchslogik wurde durchschnittlich 148 579 Zyklen, somit ein ungefährer Mehraufwand 1 % für diesen Fall gemessen.

# 6 Bewertung

Für generische Systemaufrufe konnte ein Modul für den *Linux*-Kern entwickelt werden, um ein Wiederversuchen von Systemaufrufen zu ermöglichen. Um den vorgestellten Fehlertoleranzmechanismus zu testen, wurde eine Anwendung entwickelt, die Wettlaufbedingungen erzeugt, die mit dem vorgestellten Mechanismus toleriert werden können. Wie bei vielen Fehlertoleranzmechanismen basiert auch hier der Einsatz auf einer individuellen Entscheidung. Verschiedene Metriken und Ideen können als Basis dienen, um ein spezifisches Fehlermodell zu erstellen und daraufhin die Implementierung für ein System vorzunehmen. Eine Anwendung und das zugehörige Fehlermodell konnte als Beispiel und Test herangezogen werden. Für diese Anwendung konnte gezeigt werden, dass das Wiederversuchen von Systemaufrufen eine effektive Methode bereitstellt, die erzeugte Wettlaufbedingung zu umgehen und die Anwendung verlässlich auszuführen. Der Mehraufwand für eine einfache Umgebung ohne Fehlerzustände konnte auf 1 % beziffert werden. Konzepte aus dieser Arbeit können eingesetzt werden, wenn der Anwendungsbereich spezifisch und die Fehlzustände innerhalb eines Systemaufrufs absehbar und ohne Nebeneffekte sind. Dafür können alle Vorkommen eines Systemaufrufs auf die gleiche Art und Weise behandelt werden - ein „Vergessen" einer Zeile ist nicht möglich und somit auch für andere Bibliotheken gültig. Andererseits kann dies jedoch auch zu Problemen führen, wenn Wiederversuchsheuristiken verschiedenes Verhalten innerhalb einer Umgebung, beispielsweise bei verschiedenen verbundenen Geräten erfordern. *LD_PRELOAD* kann hingegen nur die Standardbibliothek abdecken, was aber in den meisten Fällen ausreichen dürfte. Auch lassen sich die beschriebenen Mechanismen in die Ausführungsumgebung einbauen und sind somit jederzeit nachrüstbar und erfordern keine Modifikation der Anwendung. Wenn die hier beschriebenen Mechanismen der Fehlertoleranz und somit der Verlässlichkeit dienlich sind, bieten sie keine Rechtfertigung, Rückgabewerte nicht zu behandeln, sondern lediglich den immergleichen Wiederversuchsproxy uniform und konsequent umzusetzen.

Die Quellen sind auf GitHub[5] verfügbar.

[5] `https://github.com/fidoriel/Fault-Tolerant-Syscalls-Patcher`

## Literaturverzeichnis

[1] K. Bhat, E. v. d. Kouwe, H. Bos und C. Giuffrida. „FIRestarter: Practical Software Crash Recovery with Targeted Library-level Fault Injection“. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2021, Seiten 363–375. DOI: 10.1109/DSN48987.2021.00048.

[2] D. Cotroneo, A. K. Iannillo, R. Natella und S. Rosiello. „Dependability Assessment of the Android OS Through Fault Injection“. In: *IEEE Transactions on Reliability* 70.1 (2021), Seiten 346–361. DOI: 10.1109/TR.2019.2954384.

[3] *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide, Part 1*. [Online; accessed 3. Feb. 2025]. Sep. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[4] *Kernel Randomized Faulter*. [Online; accessed 27. Jan. 2025]. Jan. 2025. URL: https://github.com/trailofbits/krf.

[5] T. Naughton, G. Vallée, C. Engelmann und S. L. Scott. „A Case for Virtual Machine Based Fault Injection in a High-Performance Computing Environment“. In: *Euro-Par 2011: Parallel Processing Workshops*. Berlin, Germany: Springer, 2012, Seiten 234–243. ISBN: 978-3-642-29737-3. DOI: 10.1007/978-3-642-29737-3_27.

[6] *[N/U,04/11] x86/syscall: Don't force use of indirect calls for system calls - Patchwork*. [Online; accessed 3. Feb. 2025]. Feb. 2025. URL: https://patchwork.ozlabs.org/project/ubuntu-kernel/patch/20240411063027.493165-5-andrea.righi@canonical.com/#3296251.

[7] *[PATCH 3/3] kallsyms: Unexport kallsyms_lookup_name() and kallsyms_on_each_symbol() - Will Deacon*. [Online; accessed 3. Feb. 2025]. Feb. 2025. URL: https://lore.kernel.org/lkml/20200221114404.14641-4-will@kernel.org.

[8] G. Portokalidis und A. D. Keromytis. „REASSURE: A Self-contained Mechanism for Healing Software Using Rescue Points“. In: *Advances in Information and Computer Security*. Berlin, Germany: Springer, 2011, Seiten 16–32. ISBN: 978-3-642-25141-2. DOI: 10.1007/978-3-642-25141-2_2.

[9] F. Qin, J. Tucek, J. Sundaresan und Y. Zhou. „Rx: treating bugs as allergies—a safe method to survive software failures“. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: Association for Computing Machinery, 2005, 235–248. ISBN: 1595930795. DOI: 10.1145/1095810.1095833.

[10] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh und A. D. Keromytis. „ASSURE: automatic software self-healing using rescue points“. In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA:

Association for Computing Machinery, 2009, 37–48. ISBN: 9781605584065. DOI: 10.1145/1508244.1508250.

[11] *x86/asm: Pin sensitive CR0 bits - kernel/git/torvalds/linux.git - Linux kernel source tree*. [Online; accessed 3. Feb. 2025]. Feb. 2025. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8dbec27a242cd3e2816eeb98d3237b9f57cf6232.

[12] *x86/asm: Pin sensitive CR4 bits - kernel/git/torvalds/linux.git - Linux kernel source tree*. [Online; accessed 3. Feb. 2025]. Feb. 2025. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=873d50d58f67ef15d2777b5e7f7a5268bb1fbae2.