seL4 -
Ein sicherer Mikrokernel

B.Sc. Benjamin Zaiser
Hochschule der Medien
Nobelstr. 10, 70569 Stuttgart
E-Mail: bz003@hdm-stuttgart.de

Abstract

Frozen computer systems, viruses or trojan horses are known by nearly all people who have to work with computers. These problems are possible because of bugs in the operating systems which also lead to security vulnerabilities. To avoid these bugs in the development cycle, the kernel of the system has to be as small and simple as possible. Furthermore the isolation between components has to be improved. Microkernels fullfill these requirements. Especially the seL4 Microkernel implements a security improved API and it is really small. Because of that, it is possible to verify the whole kernel code by a so called theorem proofer. The seL4 kernel is a 100% bug-free kernel. This is achieved by an optimized inter-prozess communication and a sophisticated memory management technique.

1 Kurzfassung

Eingefrorene Computersysteme oder Viren und Trojaner sind mittlerweile fast jedem ein Begriff. Diese Probleme sind möglich, da der Kern eines Betriebssystems Fehler enthält, die zu einer Beeinträchtigung der Systemsicherheit führen können. Um diese Fehler in der Entwicklung zu vermeiden, muss die Software, die die Trusted Computing Base (TCB) [1] darstellt (hier der Kernel), möglichst klein gehalten werden. Durch die Isolation der einzelnen Komponenten wird die Sicherheit zusätzlich erhöht. Mikrokernel erfüllen diese Anforderungen. Besonders der seL4 Mikrokernel implementiert eine sicherheitsoptimierte API. Außerdem ist er sehr klein gehalten. Dies ermöglicht, dass der gesamte Code des Kernels mithilfe von sogenannten Theorem Proofern verfiziert werden kann. Das seL4 Projekt bietet einen Kernel an, der zu 100% fehlerfrei ist. Erreicht wird dies durch optimierte Inter-Prozess Kommunikation und eine ausgefeilte Memory Management Technik.

[1] Die TCB besteht aus Hardware, Firmware und Software Komponenten, welche sicherheitskritisch sind. Befindet sich innerhalb der TCB ein Fehler, so kann dieser Auswirkungen haben, die das gesamte System betreffen. Gibt es im Gegensatz dazu Fehler in Software, die sich außerhalb der TCB befindet, wird nicht das ganze System beeinträchtigt oder instabil [31].

2 Einleitung

Den bekannten „Bluescreen of Death“ von Windows oder die Linux Kernel Panic kennt mittlerweile fast jeder, der sich intensiv mit Computersystemen auseinandersetzt. Aber auch zu Zeiten von Microsoft Windows 98 war ein Bluescreen keine Seltenheit (man denke an die öffentliche Präsentation von Windows 98, gehalten von Bill Gates).

Diese Probleme entstehen, wenn Software, insbesondere der Kernel des Betriebssystems, Fehler enthält. In ungefähr 10.000 lines of code (LOC) befindet sich auch bei noch so sauberer Programmierung mindestens ein Fehler. Diese Fehler können zum einen das oben genannte Einfrieren des Systems verursachen, sie bieten aber auch offene Tore für Personen mit der Absicht, ein System zu übernehmen bzw. vollen Zugriff auf ein System zu bekommen.

Eine Studie [27] hat nachgewiesen, dass sich im Apache Webserver (229.000 LOC) zwei Bugs befinden, die aktuell ausgenutzt werden könnten, um das System zu kompromittieren. In OpenSSH (59.000 LOC) befinden sich sogar fünf solche Fehler. Bei einer gesamten Linux Distribution (ca. 60.000.000 LOC) sammeln sich so 108 echte Bugs an, die von böswilligen Personen ausgenutzt werden könnten. Bei der Prüfung wurde der Code auf bekannte Muster von Fehlerquellen hin untersucht (wie z.B. TOCTTOU, Temporary Files, Chroot Jails, ...) [3][12].

Besonders in Embedded-Systemen nimmt die Bedeutung der Sicherheit immer mehr zu. PDAs oder Handys sind mittlerweile ein integraler Bestandteil unseres Lebens. Sie werden auch immer mehr mit hoch-sensitiven Daten versorgt, wie z.B. beim eBanking vom Handy aus, E-Mails, Adressen, Notizen, usw. Bei Fehlern im Betriebssystem des PDA / Handy sind wesentlich mehr Menschen betroffen, als beim PC - laut Informa Telecoms & Media gibt es weltweit 3,3 Milliarden Handy-Verträge (2007) [20] aber laut eTForecasts nur ca. 1,2 Milliarden Internet-Nutzer (2006) [13].

Eine weitere Besonderheit der mobilen Systeme ist ihre drahtlose Kommunikation. Eine böswillige Person muss keinen direkten (physikalischen) Zugang zum Gerät haben und das Gerät an sich muss keine Internet-Verbindung besitzen. Durch Bluetooth könnte sich die Person z.B. im selben Zugabteil befinden und das Gerät kompromittieren. Immer mehr werden auch low-level Funktionen der mobilen Geräte softwaretechnisch gelöst - ein Handy-Virus könnte somit z.B. durch Aussenden eines Jam-Signals das Mobilfunk-Netz stören.

Die eingesetzte Software in Embedded Systemen wird außerdem immer komplexer. Mittlerweile werden sogar Betriebssysteme eingesetzt, die für dieses Umfeld überhaupt nicht geschaffen wurden (wie z.B. Linux, ...). Mobile Systeme verwandeln sich von Single-Vendor Closed Systems zu offenen Systemen mit Software von Dritt-Anbietern, welche potentielle Fehler enthalten könnte, die eine Sicherheitsgefahr für das gesamte System darstellen. Die mobilen Systeme sind meist Safety-Critical[2] oder Mission-Critical[3] (z.B. Headunit in Oberklasse-Fahrzeugen) und dem Thema Sicherheit muss hierbei besondere Aufmerksamkeit gewidmet werden.

2.1 Motivation

Wie schafft man es nun, die immer komplexer werdenden Systeme zu vereinfachen und wie schafft man ein Betriebssystem, das fehlerfrei ist?

Um die Komplexität des Systems möglichst gering zu halten, müssen Isolationstechniken verwendet werden, wie z.B. Prozesse oder Virtual Memory. Durch die Unterteilung in einzelne Komponenten wird die Isolation untereinander verstärkt - ein Fehler in einer Komponente kann sich somit nicht auf andere Teile des Systems ausbreiten.

Um die Fehler aus dem Betriebssystem bzw. dem Kernel zu entfernen, könnten folgende Methoden verwendet werden: Testing, Code Inspection oder Model Checking[4]. Durch diese Methoden können aber nicht *alle* Fehler gefunden werden. Zum Beispiel beim Testing: Dijkstra hat einmal gesagt, „Testing can only show the presence, not the abscence, of bugs“ [15]. Die einzige Möglichkeit, einen Kernel fehlerfrei zu machen, ist die komplette Verifikation des Codes hinsichtlich seiner Spezifikation. Dies ist eine Mammut-Aufgabe, da ein Mikrokernel zwischen 7.000 - 10.000 LOC enthält. Die Trusted Computing Base, hier der Kernel, muss somit möglichst gering gehalten werden. Nur so wird eine Verifizierung des Codes überhaupt erst möglich. Im Falle von großen monolithischen Kerneln, die aus einigen 100.000 LOC bestehen, ist eine Verifizierung durch die Größe und Komplexität des Systems unmöglich (Der Linux-Kernel besteht aus ca. 4.1 Millionen LOC; der Kernel von Windows Vista wird auf ungefährt 20 Millionen LOC geschätzt [16]).

[2]Hat ein Safety-Critical System einen Fehler, so könnte dieser folgende Konsequenzen haben: Tod oder ernsthafte Verletzungen von Menschen; Verlust oder ernsthafter Schaden des Geräts; Beeinträchtigung der Umwelt.

[3]Hat ein Mission-Critical System einen Fehler, so könnte der Projekterfolg bzw. der Einsatzzweck des Systems beeinträchtigt oder gar nicht erreicht werden.

[4]Formale Methoden, die nachweisen können, dass bestimmte Kategorien an Fehlerquellen nicht vorhanden sind. Dadurch kann aber nicht nachgewiesen werden, dass es keine Fehler gibt.

2.2 Aufbau des Dokuments

Zunächst wird ganz kurz der L4 Mikrokernel beschrieben, welcher die Grundlage des seL4 Kernels bildet. Auf dem L4 Kernel bauen viele unterschiedliche Projekte auf. Einen kurzen Überblick gibt es unter Kapitel 4, um das seL4-Projekt besser einordnen zu können. Da das seL4 Projekt eng mit dem L4.verified Projekt zusammenhängt, wird dieses in einem eigenen Abschnitt beschrieben. Der Hauptteil befasst sich intensiv mit dem seL4 Kernel. Angefangen bei den Kernel-Objekten, über Threads, das verwendete Access Control Model, das Memory Management bis hin zur Inter-Prozess Kommunikation. Abschließend wird ein kurzes Fazit gezogen und der aktuelle Stand des Projekts erläutert.

3 Der L4 Mikrokernel

Anfang der 90er Jahre wurde einer der ersten Mikrokernel entworfen, der so genannte Mach Kernel. Er wurde entwickelt um den UNIX Kernel in der BSD Version zu ersetzen. Heutzutage wird er nicht mehr weiterentwickelt. Allerdings ist er oder ein Derivat noch in diversen Betriebssystemen im Einsatz (z.B. NeXTSTep oder als wesentlich verbesserte/erweiterte Version in Mac OS X). Das große Problem des Mach Kernels ist seine schlechte Performance, verursacht durch eine zu komplizierte Inter-Prozess Kommunikation. Jochen Liedtke bewies Mitte der 90er Jahre, dass durch eine gut designte Inter-Prozess Kommunikation ein Mikrokernel dazu fähig ist, Real-Time Systemanforderungen zu erfüllen. Um den hohen Performance-Ansprüchen gerecht zu werden, wurde der L4 Mikrokernel komplett in Assembler geschrieben. Die API des L4 Kernels wurde mittlerweile von vielen verschiedenen Projekten reimplementiert und weiterentwickelt im Hinblick auf Platformunabhängigkeit, Security, Isolation und Robustheit [29][30].

4 Überblick der Projekte, die an dem L4 Mikrokernel arbeiten

Außer den Projekten seL4 und L4.verified, die später ausführlich erläutert werden, gibt es noch weitere Projekte, die mit dem L4 Mikrokernel arbeiten [18].

L4Ka::Pistachio Dies ist der aktuellste L4 Mikrokernel, der die L4 Version 4 API implementiert. Diese API ist 32- und 64-Bit fähig, beinhaltet Multiprozessor-Unterstützung und eine sehr schnelle Inter-Prozess Kommunikation. Das Projekt wird geleitet von der System Architecture Group an der Universität Karlsruhe in Zusammenarbeit mit der DiSy group an der Universität von South Wales. In den Quellcode flossen sieben Jahre Forschung an Mikrokernen und Multi-Server Systemen ein. Er wurde in C++ geschrieben mit Fokus auf Performance und Portabilität. Vorhergehende Projekte wie z.B. L4Ka::Hazelnut oder L4/Alpha wurden durch dieses Projekt abgelöst [17].

NICTA::Pistachio-embedded NICTA[5]::Pistachio-embedded ist eine Weiterentwicklung des L4Ka::Pistachio Projekts. Dabei wurde der L4 Kernel für den Embedded-Bereich optimiert. Insbesondere die Kernel-Komplexität und der Memory Footprint wurden reduziert. Das Projekt wurde von OKL4 abgelöst und wird nicht mehr fortgesetzt.

OKL4 Das OKL4 Projekt ist der Nachfolger des NICTA::Pistachio-embedded Projekts, welches auf dem L4Ka::Pistachio Mikrokernel basiert. OKL4 ist ein kommerzielles Projekt der Open Kernel Labs. OKL4 unterstützt sehr viele Architekturen und kann vom Embedded-Bereich bis hin zu Multi-Prozessor Enterprise Systemen eingesetzt werden. Die Open Kernel Labs arbeiten eng mit dem NICTA Forschungszentrum zusammen. So fließen die Erkenntnisse aus der Forschung direkt in das Produkt ein.

Fiasco Dies ist ein verbesserter Mikrokernel für x86 Systeme entwickelt von der Technischen Universität Dresden. Er implementiert die L4 API und ist somit auch kompatibel mit L4/x86 entwickelt von Jochen Liedtke. Geschrieben in C++ bietet er bessere Echzeit-Eigenschaften als der L4 Kernel [28].

L^{4}Linux L^{4}Linux ist eine Portierung des Linux Kernels auf die L4 Mikrokernel API. Dadurch ist es möglich, Linux im User-Mode auf einem Mikrokernel (z.B. Fiasco) Seite an Seite mit anderen Mikrokernel Applikationen, wie z.B. einer Echtzeit-Komponente, auszuführen [25].

DROPS Das Dresden Real-Time Operating System ist ein Betriebssystem, das gleichzeitig Real-Time und Time-Sharing Applikationen ausführen kann. Es basiert auf dem Fiasco Mikrokernel und verwendet unter anderem L^{4}Linux [7].

[5]NICTA ist ein Forschungszentrum für Information und Kommunikation in Sydney, Australien. Besonders die Forschungsabteilung ERTOS (Embedded Real-Time Operating Systems) beschäftigt sich mit der Entwicklung von Mikrokernen.

Potoroo Probabilistic Temporal Analysis of Operating Systems Code. Dieses Projekt hat die Zielsetzung, ein Analyse-Model zur Verfügung zu stellen, welches das zeitliche Verhalten des L4 Kernels anschaulich darstellt. Das Ergebnis ist dabei ein worst-case Ausführungszeit-Profil für alle System Calls. Entwickelt wird es am NICTA Forschungszentrum [24].

Iguana Iguana baut auf dem seL4 Kernel auf und bietet somit die Basis für die Bereitstellung von Services, die ein Betriebssystem benötigt (Memory und Protection Management, Gerätetreiber-Framework, ...). Iguana beachtet dabei die neuen Sicherheitsfeatures des seL4 Kernels.

Wombat Auf Iguana baut das Wombat Projekt auf. Wombat ist eine Version eines paravirtualisierten Linux Systems, um Embedded Systemen Legacy Support zu geben [21].

5 L4.verified

Das L4.verified Projekt hängt eng mit dem seL4 Projekt zusammen. Es hat die Aufgabe, den gesamten L4 Kernel mathematisch auf seine funktionale Korrektheit zu verifizieren, basierend auf einer High-level formalen Beschreibung. Ziel ist es, einen absolut vertrauenswürdigen hochperformanten Mikrokernel herzustellen. Dieses Projekt wird ebenfalls vom NICTA Forschungszentrum durchgeführt.

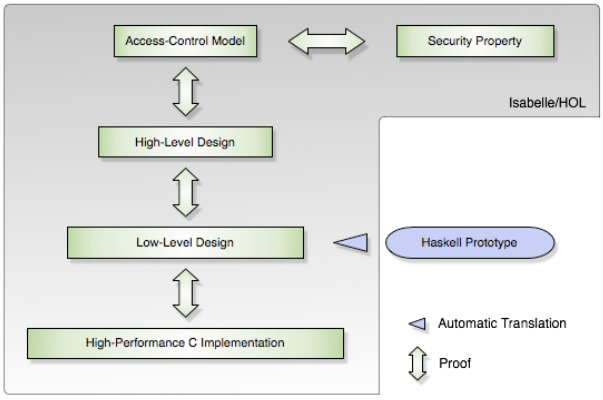

Abbildung 1: Aufbau L4.verified [23]

Als Herangehensweise wird die formale Verfeinerung angewandt. Auf der untersten Ebene befindet sich dabei eine hochperformante C / Assembler Implementierung des seL4 Kernels. Eine Ebene höher befindet sich eine ausführbare Spezifikation des Kernel Prototyps, der aus dem seL4 Projekt hervorgeht. Der Prototyp wurde dabei in der funktionalen Sprache Haskell geschrieben, mehr dazu unter 6.1. Auf der nächsten Ebene befindet sich eine abstrakte, weniger detaillierte Spezifikation des Kernels. Auf der obers-

ten Ebene befindet sich ein abstraktes Access Control Model des seL4 Kernels, welches die Verteilung der Capabilities im System darstellt. Jede Verfeinerungsebene muss verifiziert werden. Dies geschieht mithilfe von Isabelle/HOL [23].

Isabelle ist ein Theorem-Proofer. Die Software erlaubt mathematische Formeln in einer formalen Sprache auszudrücken und bietet zusätzlich ein Werkzeug an, das diese Formeln überprüfen kann. Es kann eingesetzt werden um mathematische Sätze zu beweisen, Schaltkreise zu überprüfen oder ganze Programme zu verifizieren. Entwickelt wird es von Larry Paulson, Universität von Cambridge und Tobias Nipkow, Technische Universität München [2]. HOL ist eine Entwicklungsumgebung, um ein Theorem interaktiv in einer Higher-Order Logic[6] zu beweisen. HOL kann durch eine Meta-Sprache (ML) programmiert werden und ist somit universell einsetzbar [1].

Das L4.verified Projekt soll bis Ende 2008 fertiggestellt sein. Bis jetzt wurden jedoch bereits einige Teilergebnisse veröffentlicht.

- Durch das Beenden der Verifizierung von dem High-Level und Low-Level Design, ist der seL4 Kernel der beste und am tiefsten analysierte High-Performance Kernel der Welt.
- In Zusammenarbeit mit dem seL4 Projekt wurde eine Methode und die entsprechenden Werkzeuge entwickelt für das Rapid-Prototyping eines kleinen Betriebssystem-Kernels. Diese Methode hat kurze Zyklen, die es ermöglichen, schnell neue Features auszuprobieren. So benötigt das Einbauen neuer Funktionen nur Stunden oder Tage anstatt Wochen oder Monate.
- Es wurde eine präzise Formalisierung der Programmiersprache C und deren Memory Model entworfen.

6 seL4

Der seL4 Mikrokernel (Secure Embedded L4) ist eine Erweiterung des L4 Mikrokernel bzw. baut auf diesem auf. Die wichtigste Erweiterung ist, dass Rechte durch Capabilities vergeben werden. Kein Objekt besitzt standardmäßig geerbte Autorität. Jegliche Autorität muss durch eine Capability zugewiesen werden. Außerdem besitzt seL4 ein explizites Speichermanagement, bei dem Kernel-Datenstrukturen von den Applikationen direkt allokiert werden können, indem sie die nötige Autorität zugewiesen bekommen. Eine weitere Neuerung gibt es in der Kernel-Entwicklungsmethodik. SeL4 wird nicht in C designed und implementiert (was sehr zeitaufwendig sein kann) sondern in Haskell. Mit der Zeit hat sich die Haskell-Implementation von einem einfachen Modell zu einem kompletten Kernel entwickelt. Der Haskell-Code kann auf einem speziellen Simulator ausgeführt werden, um den Kernel zu testen [16].

6.1 Anforderungen an das System

Das seL4 Projekt ist in zwei Aufgaben unterteilt. Zum einen soll die API des L4 Mikrokernels in puncto Sicherheit verbessert werden und zum andern soll die Mikrokernel API spezifiziert, entwickelt und validiert werden. Das Projekt wird vom NICTA Forschungszentrum durchgeführt. Mehr dazu unter 6.1 [22].

Die sicherheitstechnisch zu verbessernden Bereiche der L4 API ist zum einen die Inter-Prozess-Kommunikation (IPC) und zum anderern das physikalische Speicher-Management. Die IPC muss sicherstellten, dass die Isolation der Komponenten durch deren Kommunikation nicht durchbrochen werden kann. Das Speicher-Management ist dafür verantwortlich, die Kernel Services bereitzustellen, sowie deren Ausführungszeit vorhersagbar zu machen.

Folgende Ziele wurden für das seL4 Projekt gesetzt:

- Die API muss präzise sein. Eine wörtliche Beschreibung ist unzufriedenstellend und zweideutig.
- Es müssen Beispiel-Implementierungen angefertigt werden, die den Leser überzeugen, dass der Kernel hochperformant sein kann.
- Es soll eine Methode für die Entwicklung von Higher-level Systems zur Verfügung gestellt werden.
- Die Herangehensweise muss auch für Kernel-Entwickler nachvollziehbar sein, die sich nicht in formalen Methoden auskennen.

Um diese Ziele zu erreichen, wurde für die Spezifikation und die Implementierung die formale Sprache Haskell [7] verwendet. Haskell eignet sich dafür besonders gut, da die Sprache keine Seiteneffekte besitzt, automatisch für Isabelle/HOL übersetzt werden kann, ein ausführbares Modell des Kernels bietet und von den meisten Kernel-Programmierern nachvollzogen werden kann [4].

Für die Validierung des Systems wurde ein Simulator erstellt, welcher einen realen Kernel emuliert.

[6] Die „Logik höherer Stufe“ ist eine Erweiterung der Prädikatenlogik und basiert auf dem typisiertem Lambda-Kalkül. Die Prädikatenlogik untersucht nicht nur die Aussage sonder auch deren innere Struktur.

[7] Haskell ist eine rein formale Programmiersprache welche auf dem Lambda-Kalkül basiert. Das bedeutet, dass Funktionen nur Werte zurückgeben, nicht aber den Zustand des Programms ändern. Es entstehen also keine Seiteneffekte. Das Ergebnis der Methode hängt stets von den Eingabeparametern ab und für dieselben Eingabeparameter wird immer (egal in welchem Zustand sich das System befindet) dasselbe Ergebnis zurückgegeben [26].

6.2 Kernel-Objekte

Die verschiedenen Aufgaben eines Kernels werden mithilfe von so genannten First Class Kernel Objects abstrahiert. Jedes Objekt bietet dabei verschiedene Methoden an [8][10].

CNode Enthält ein Array an Capabilities mit der Größe $2^n (n > 0)$. Ein CNode ist sozusagen ein Wrapper um die Capability Objekte. Ein Zugriff auf eine Capability kann *nur* durch ein CNode erfolgen. Durch den CNode kann dann geprüft werden, ob der Zugriff auf eine Capability rechtmäßig/zulässig ist.

TCB Ein TCB (Thread Control Block) Objekt abstrahiert einen Thread. Das Erzeugen eines Threads entspricht der Instantiierung eines TCB Objekts. Das TCB Objekt enthält die Konfiguration und den Kontext des Threads.

Endpoint Abstrahiert die IPC. Threads senden und empfangen Nachrichten, indem sie eine Capability an Endpoints aufrufen. Das Objekt enthält intern eine Queue, in der sich die wartenden Nachrichten befinden. Zusätzlich enthält das Objekt noch ein Flag, das angibt, ob die in der Queue befindlichen Nachrichten auf eine send oder wait Operation warten.

Asynchronous Endpoint Diese Objekte dienen zur asynchronen IPC. Im Gegensatz zu den normalen Endpoints enthalten diese Objekte keine Queue sondern einen Puffer. Dieser wird verwendet um den Inhalt einer Nachricht zu speichern, nachdem der Sender seine Verarbeitung wieder aufgenommen hat.

Untyped Memory Aus diesem Objekt können alle anderen Kernel Objekte instantiiert werden. Der gesamte Kernel Speicher besteht beim Systemstart aus Untyped Memory Objekten. Mehr dazu unter 6.5.

User Data User Data Objekte haben eine von Größe: 2^n, die vom User her zugänglich sind. Load und Store Befehle entsprechen einem Aufruf dieser Objekte.

Weitere wichtige Begriffe im Bezug auf die Kernel Objekte:

Capability Eine Capability entspricht einer Referenz auf ein Kernel-Objekt zuzüglich einem Set an Zugriffsregeln über dieses Objekt. Da durch eine Capability Autorität übertragen wird, muss diese unveränderbar sein. Eine Veränderung ist nur möglich, indem die Capability entfernt und durch eine andere ersetzt wird. Der Besitz einer Capability ist eine nötige und hinreichende Bedingung, um Operationen auf einem Kernel Objekt durchzuführen.

CSpace Ein CSpace (Capability Address Space) ist eine hierarchische Kollektion von CNode Objekten. Jeder Thread besitzt einen eigenen CSpace. Dieser dient als Namespace, da jeder Thread streng von einem anderen getrennt sein muss. Ein Thread kann auf alle Objekte zugreifen, die von der Wurzel seines CSpace erreichbar sind.

System Call Ein System Call entspricht dem Aufruf einer Capability. Die Argumente bestehen aus Daten und anderen Capabilities, die erforderlich für den System Call sind.

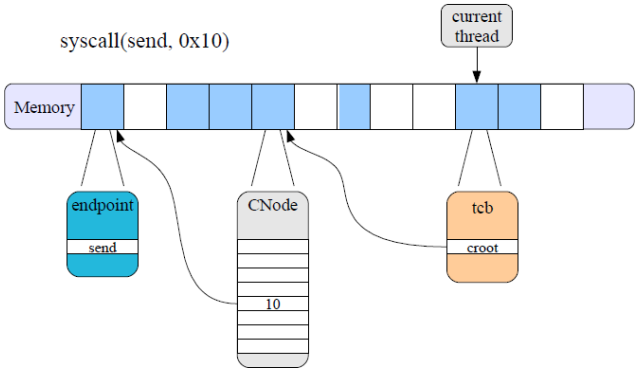

Abbildung 2: Aufruf eines Systemcalls [10]

Ein System Call läuft wie folgt ab: z.B. *syscall(send, 0x10)* (Operation: send; erforderliche Capability / bzw. das Kernel Objekt auf dem die Operation ausgeführt werden soll: 0x10) wird vom Kernel empfangen. Danach wird die Wurzel (croot) des CSpace lokalisiert, der mit dem aufrufenden Thread verbunden ist. Die Wurzel des CSpace, die im TCB Objekt gespeichert ist, entspricht einem Zeiger, der auf ein CNode Objekt verweist (siehe 6.2). Das CNode Objekt „übersetzt“ nun sozusagen die Capability in einen Zeiger auf ein anderes CNode Objekt oder auf ein Kernel Objekt, wie z.B. einem Endpoint. Bei jedem Aufruf wird überprüft, ob die Capability auch genügend autorität besitzt, um die angeforderte Operation auszuführen. Wenn dem so ist, dann wird das Objekt auf das die Capability verweist mit den restlichen Argumenten aufgerufen. Ansonsten benachrichtigt der Kernel einen designierten Thread oder die Operation wird stillschweigend verworfen.

6.3 Threads

Jeder User-Level Thread besitzt ein TCB Objekt. Ein neuer Thread wird erzeugt, indem ein Untyped Memory Objekt durch die retype Operation in ein TCB Objekt verwandelt wird. Für die Initialisierung und die Kontrolle werden zwei zusätzliche System Calls benötigt. Diese entsprechen den

System Calls, wie sie im L4Ka::Pistachio Projekt bereits definiert wurden [6].

ThreadControl Dieser System Call setzt die verschiedenen Konfigurationsparameter eines Threads. Er wird aufgerufen bevor ein Thread gestartet wird, kann aber auch auf einen laufenden Thread ausgeführt werden, um die bestehende Konfiguration zu ändern. Folgende Parameter können konfiguriert werden:

- Thread Priorität
- Capability zum Root des CSpace
- Capability zum Default Page Fault Handler Endpoint
- Capability zu speziellen Page Fault Handlern

ExchangeRegisters Dieser System Call erlaubt einem User-Level Server Thread, seine Clienten zu sichern oder wiederherzustellen. Der Aufruf gibt direkten Zugang zu dem Kontext anderer Threads, indem der gesamte Kontext oder nur ein Teil zwischen zwei explizit spezifizierten Threads kopiert wird.

6.3.1 Scheduler

In seL4 wird ein Multipriority Round Robin Scheduler verwendet. Dieser besitzt separate Queues von Runnable Threads für jede Priorität. Die Priorität und die Ausführungszeit kann jedem Thread individuell zugewiesen werden. Dadurch ist es möglich, verschiedene Standard-Scheduler zu emulieren, wie z.B. einen Rate-Monotonic Scheduler (eindeutige Prioritäten mit unendlichen Time Slices) oder einen Proportional-Share Scheduler (eine Priorität mit unterschiedlich langen Time Slices) [11].

6.4 Access Control Models

Ein Access Control Model stellt ein Framework bereit, um Security Policies zu spezifizieren, zu analysieren und zu implementieren. Das Modell besteht aus einem begrenzten Set an Zugriffsrechten und einem begrenzten Set an Regeln, um diese Reche zu modifizieren oder zu verteilen. Eine Sicherheitsanalyse des Modells würde wie folgt ablaufen: zunächst wird ein gegebenes Set an Regeln und eine festgelegte Verteilung an Zugriffsrechten im System bestimmt. Nun wird geprüft, ob es möglich ist, einen bestimmten Systemstatus zu erreichen, bei dem ein Objekt ein bestimmtes Zugriffsrecht bekommt, das es zu Beginn noch nicht hatte. Das klassische Access Control Model in Capability Systemen ist das Take-Grant Model [10].

6.4.1 Take-Grant Model

Das Take-Grant Model gibt eine Aussage darüber, wie Rechte und Informationen in einem System fließen können. Es kann anschaulich als Graph dargestellt werden. Das Model entspricht einem Tripel <S, R, O>. S ist ein Subjekt (aktiv), O ist ein Objekt (passiv), R sind Rechte bzw. Regeln wie z.B. read (r), write (w), create, remove, grant, take, etc. Subjekte und Objekte bilden die Knoten in dem Graph. Rechte werden in Form von Capabilities in den Knoten selbst abgelegt. Im Graph entsprechen die Kantenbezeichnungen, der Autorität der Knoten zueinander.

- Besitzt ein Knoten A ein take Recht an Knoten B, so darf A sich Rechte aus B nehmen.
- Besitzt ein Knoten A ein grant Recht an Knoten B, so darf A eigene Rechte in B ablegen.
- Mit der create Regel darf ein Knoten A einen neuen Knoten B erstellen. Beide Knoten sind dabei mit einer Kante mit der Bezeichnung α verbunden.
- Mit der remove Regel kann ein Knoten entfernt werden. Dabei wird zunächst die Bezeichnung der Kante entfernt und wenn $\alpha - \beta = \{\}$ gilt, wird die Kante an sich entfernt.

Bei dem Take-Grant Modell ist zu beachten, dass es auch zu indirekten Rechten kommen kann. Hat z.B. ein Objekt A das Recht w auf Objekt B (de facto) und B das Recht w auf C (de facto), so besitzt A ein indirektes Recht w auf C (de jure) [14].

In seL4 wird dieses Modell leicht verändert angewendet. Aufgrund der Restriktion, dass das Recht, neue Objekte zu erstellen von den Untyped Memory Objects ausgeht (siehe 6.5.2), ist die create Regel nur anwendbar, wenn der Ersteller auch eine create Authority besitzt. Ebenfalls wird die remove Regel modifiziert: Anstatt zuerst die Bezeichnung der Kante zu entfernen, wird gleich die ganze Kante entfernt. Diese Modifikationen sind möglich, da im seL4 Kernel die Capabilities unveränderbar sind. Zusätzlich wird in seL4 eine zusätzliche Regel definiert, um eine einzelne Capability zu entfernen: revoke (mehr dazu unter 6.5.4). Die take Regel wird in seL4 nicht angewandt, da die Weitergabe von Rechten stets mithilfe der Grant Methode gelöst wird [10].

6.5 Memory-Management

6.5.1 Aufbau

Außer einem kleinen Bereich von statisch allokiertem Speicher für den Kernel-Code und den Kernel-Stack wird der gesamte Speicher von User-Level Tasks verwaltet. Das wird erreicht, indem diesen Tasks Capabilites zu diversen Speicherbereichen gegeben werden. So werden DoS-Attacken verhindert, da jeder Thread explizit angeben muss,

wieviel Speicher er allokieren möchte. Dabei kann überwacht werden, wie viel Speicher ein Thread bereits besitzt und unter Umständen kann eine weitere Speicherallokierung verhindert werden [6].

Beim Startvorgang wird zunächst der Speicherbereich für den Kernel Code / Stack allokiert. Der gesamte restliche Speicher wird in Untyped Memory Objects gepackt. Der initiale Thread ist der Ressource Manager, welcher die Untyped Memory Objects verwaltet/besitzt. Der Ressource Manager hat außerdem die Aufgabe, das restliche System zu bootstrappen (entspricht dem init Prozess bei Linux Systemen) mehr zum Ressource Manager unter 6.5.3 [10].

6.5.2 retype-Operation

Untyped Memory Objekte können mithilfe der retype Operation in kleinere Untyped Memory Objekte oder in andere Kernel Objekte verwandelt/verfeinert werden. Damit die Integrität der Kernel Objekte gewährleistet werden kann, darf ein Speicherbereich stets nur ein einziges Kernel Objekt beinhalten. Deswegen enthält die retype Operation folgende Restriktionen:

- Die Child Objekte dürfen sich nicht überlappen - das durch die retype Operation erzeugte Objekt muss kleiner oder gleich groß sein, wie das Untyped Memory Objekt.

- Die Child Objekte müssen eindeutig sein. Das bedeutet, dass das Untyped Memory Objekt gleichzeitig keine anderen zuvor „verfeinerte“ Child Objekte beinhalten darf (In einem Untyped Memory Objekt darf sich nur ein Child Objekt befinden).

Sobald ein Kernel Objekt mithilfe der retype Operation erstellt wurde, hat der Ersteller volle Autorität über das Objekt. Das bedeutet, dass er die Methoden des Objekts aufrufen und alle oder nur eine Teilmenge der Capabilities des Objekts an seine Clients weitergeben kann (siehe Take-Grant Modell 6.4.1). Das Set von Untyped Memory Objekten, die eine Applikation besitzt, gibt den ihr verfügbaren Speicherbereich an. Die Frage der Partitionierung von Hardware Ressourcen ist also eine Frage der Capability Verteilung.

Durch diese Methodik wird sichergestellt, dass Kernel Daten von einem User-Zugriff geschützt werden, da es keine Überlappung von Objekten gibt.

6.5.3 Ressource Manager

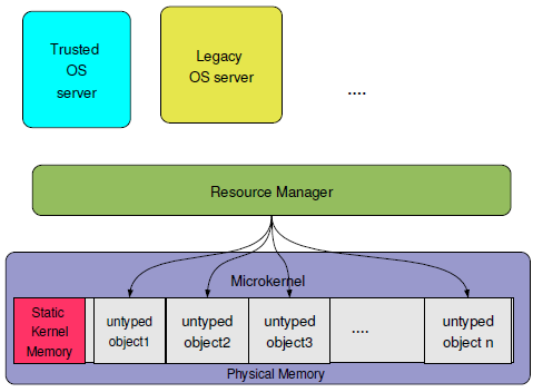

Abbildung 3: Ressource Manager [10]

Der Ressource Manager, der im Besitz aller Untyped Memory Objekte ist, kann diese nun an Gast-Betriebssysteme in Form von Capabilities weitergeben. Das Gast-Betriebssystem erhält also ein Set an Capabilities, es hat aber keine Möglichkeit, Capabilities zu bekommen, die ihm nicht zur Verfügung gestellt wurden (siehe 6.4.1). Das Interessante an dem Ressource Manager ist, dass sich dieser im User-Space befindet. Er ist nicht Teil des seL4 Kernels. Dieser muss lediglich ein sicheres Interface anbieten. So ist es dem Programmierer möglich, eine spezielle, für seinen Fall optimale Memory Management Policy zu implementieren, ohne den Kernel verändern zu müssen. Durch die Auslagerung der Policy muss nicht der Kernel entscheiden, wann, wie und wo Speicher allokiert werden muss. So kann dieser flexibler eingesetzt und zudem noch klein gehalten werden. Die auf ihm aufbauende Subsysteme könnten dadurch gleichzeitig individuelle Ressource Policies definieren.

6.5.4 Policies

Der physikalische Speicher eines Systems insbesondere bei Embedded-Systemen ist sehr begrenzt. Es wird somit ein sehr effizientes Management gefordert, damit der Speicherbedarf eines Tasks keinen anderen Task negativ beeinflusst oder sogar stoppt. Bei einem statischen System reichen einfache Quota-basierte Systeme aus. Bei einem dynamischen System muss jedoch gewährleistet werden, dass es auch bei einer Leistungsspitze den Anforderungen gerecht wird. Eine höhere Speichereffizienz kann erreicht werden, indem Speicher dynamisch neu zugewiesen wird. Gerade bei einem Real-Time System muss eine Anfrage in einer definierten Zeit beantwortet werden, egal unter welcher Last sich das System gerade befindet. Das Speichermanagement muss also eine vorhersagbare Zugriffszeit auf Daten-Strukturen liefern. Für die Verwaltung des Speichers wer-

den deshalb sogenannte Memory Management Policies definiert. Sie geben an, wie der Kernel-Speicher verwendet werden soll [9].

Speicher dynamisch zuweisen Sobald ein Speicherbereich neu zugewiesen werden soll, muss der Kernel garantieren, dass es keine noch offenen und gültigen Capabilities auf die Objekte gibt, welche sich noch in dem Speicherbereich befinden. Der seL4 Kernel löst dies auf folgende Art und Weise: die Capabilities werden in einer Baumstruktur angeordnet, dem sogenannten *Capability Derivation Tree* (CDT). Durch den CDT wird ein Referenzzähler irrelevant. Erstellt nun ein User aus einem Untyped Memory Objekt ein neues Kernel Objekt, so wird auch die damit neu erstellte Capability als Kindknoten im CDT angehängt. Bei jeder weiteren Kopie wird ein neuer Kindknoten angehängt. Damit für den CDT kein zusätzlicher Speicher allokiert werden muss, wird dieser als doubly-linked list in den Capabilities selbst gespeichert. Ist der User im Besitz des ursprünglichen Untyped Memory Objekts, so kann er die *revoke* Operation aufrufen, die alle Kindknoten entfernt. Mit dem Entfernen der letzten Capability auf das Kernel Objekt wird die *destroy* Operation aufgerufen. Diese Methode deaktiviert das Objekt und bricht damit alle internen Kernel Abhängigkeiten. Sobald die revoke Operation beendet wurde, kann dann ein neues Kernel Objekt erstellt werden.

6.6 IPC

Bei seL4 wird für die Interprozess-Kommunikation kein Partner-Thread explizit spezifiziert. Die gesamte IPC wird über spezielle Kernel Objekte, die IPC Endpoints, abgewickelt. Für die IPC gibt es zwei System Calls: *Send IPC* und *Receive IPC*. Beide Befehle werden auf ein Endpoint Objekt aufgerufen. Die Kommunikation mithilfe der Endpoints kann immer nur in eine Richtung stattfinden [5]. Sobald ein Thread den *Send IPC* Befehl auf ein Endpoint Objekt und gleichzeitig ein anderer Thread einen *Receive IPC* Befehl auf dasselbe Enpoint Objekt aufruft, initiiert der Kernel einen Message Transfer zwischen den beiden Threads. Die Nachrichten werden synchron übertragen. Das bedeutet, dass der Thread, der zuerst *Send IPC* oder *Receive IPC* aufruft, solange blockiert wird, bis die Nachricht übertragen ist. Möchten mehrere Threads gleichzeitig eine *Send IPC* Operation auf dasselbe Endpoint Objekt ausführen, so wird der Kernel nur eine Operation abarbeiten. Alle anderen Threads müssen weiterhin warten. Welche Operation der Kernel abarbeitet hängt von der Implementierung ab. Im Prototyp wird das LIFO Prinzip angewendet. Warten mehrere Threads auf den Empfang einer Nachricht, ist das Verhalten identisch. Ein Thread kann zur selben Zeit nur an einem Endpoint auf eine Nachricht warten. Wenn also ein Server Thread mehrere Clients bedient, muss jeder Client denselben Endpoint verwenden. Erhält nun der Server Thread eine Nachricht, muss ihn der Kernel zusätzlich noch darüber informieren, von welchem Client diese stammt. Deshalb bekommt jede Endpoint Capability einen Index (Badge) zugewiesen, der dem Empfänger mitgeteilt wird. Der Thread kann dabei die Zuordnung von Client zu Index festlegen [6].

Es gibt insgesamt zwei Arten der IPC. Zum einen die Übertragung von Nachrichten (Informationsfluss) und zum andern die Weitergabe von Capabilities (Autoritätsfluss oder auch Grant IPC). Beide Arten funktionieren durch die oben beschriebene Methode [9].

Bei der Weitergabe von Capabilities (Grant IPC) wird ein entsprechendes Flag in der Nachricht gesetzt. Die Nachricht selbst besteht dabei aus der Referenz zur Capability, einem Set an Capability-Rechten und optional einem neuen Capability-Datenwort. Der empfangende Thread muss eine CNode Capability-Referenz mit einem Index aus seinem CNode Objekt an den Endpoint senden. Die neue Capability-Referenz wird dann an dem angegebenen Index abgespeichert [5].

6.6.1 Isolation Domains

In seL4 werden sogenannte Isolation Domains eingeführt. Eine Isolation Domain wird definiert als eine Collection von Applikationen, deren physikalischer Speicher vom Rest des Systems isoliert ist. Applikationen innerhalb einer Isolation Domain können sich Ressourcen teilen. Das bedeutet, dass Capabilities frei verteilt werden können. Dies kann aber niemals über Isolation Domains hinweg geschehen. So ist es möglich, dass ein Guest OS Kernel sich die Ressourcen mit seinen Applikationen teilen kann. Die Mechanismen, die in seL4 angewandt werden reichen aus, um Trennung der Isolation Domains sicher zu stellen [9]. Bei diesem Verifikationsprozess unter Verwendung von Isabelle/HOL (siehe 1) konnten folgende Bedingungen/Restriktionen ermittelt werden, die bei der initialen System Konfiguration eingehalten werden müssen. Durch diese Bedingungen können die Isolation Domains außerdem für jeden zukünftigen Status des Systems gesichert werden.

- Eine Appliaktion kann nur einer Isolation Domain angehören.
- Schreibbare CNodes können nicht zwischen Applikationen in verschiedenen Isolation Domains geteilt werden.
- Kein Grant IPC zwischen zwei Applikationen in unterschiedlichen Isolation Domains.
- Eine Applikation in einer Isolation Domain darf nicht das Recht besitzen, die Autorität eines Threads in einer anderen Isolation Domain zu ändern. In anderen

Worten: die Applikation darf keine Capability zu einem TCB Objekt eines anderen Threads besitzen.

6.7 Anwendungen

Das seL4 Projekt ist noch nicht beendet (siehe 7.1). Allerdings fließen die Erkenntnisse in den OKL4 Kernel ein, der bereits kommerziell vertrieben wird. Der OKL4 Kernel wird in folgenden Bereichen mit den jeweiligen Anforderungen eingesetzt [19]:

Mobilfunk In diesem Bereich muss das System den Anforderungen vieler Beteiligten gerecht werden, wie z.B. den Mobilfunkdienstleistern, der Regierung, den End-Usern, usw. Hierbei ist es wichtig, dass die Isolation von OpenSource Code und geschütztem Code aufrechterhalten wird. Außerdem müssen trusted und untrusted Umgebungen unterstützt werden, sowie eine robuste, sichere und effiziente Kommunikation zwischen Applikationen und den Gast-Betriebssystemen. Durch den Einsatz vieler unterschiedlicher Programme von Dritten, die womöglich fehlerhaften Code enthalten, ist eine strenge Isolation der Applikationen untereinander erforderlich. Gerade bei mobilen Geräten sind die Hardware-Ressourcen sehr begrenzt. Deshalb ist ein sehr kleiner Memory Footprint erforderlich und es müssen ARM9 und ARM11 CPUs unterstützt werden.

Consumer Elektronik Im Consumer Elektronik Bereich (z.B. HiFi Anlage, TV, Heimkino, ...) muss vor allem die Hardwareuntersützung gegeben sein im Bezug auf 32 und 64 Bit SoCS, FPGAs und ARM sowie MIPS Kernen. Ebenfalls sind die Hardware Ressourcen sehr begrenzt. Durch die immer stärker werdende Vernetzung der Geräte untereinander steht hier die Sicherheit an einer hohen Stelle. Aber auch das Power-Management steht im Vordergrund. So sollte z.B. die Stand-By Stromaufnahme so gering wie möglich sein.

Netzwerkinfrastruktur Netzwerkgeräte, wie z.B. Router oder Gateways müssen hochperformant sein und die zur Verfügung stehende Hardware bestmöglich nutzen. Load Balancing und Rapid Failover ist erforderlich für eine möglichst ausfallfreie Verfügbarkeit des Systems. In diesem Bereich kommt zudem der Sicherheit des Systems größte Aufmerksamkeit zu. Netzwerkbasierte Exploits, Buffer Overflows, DoS-Attacken etc. müssen verhindert werden. Die Software muss skalieren, um auch auf Blades in einem Rack / Cluster optimale Performance zu bringen und Gastsysteme sollten im Fehlerfall schnell neugestartet werden können.

Automobil Fahrzeuge werden immer intelligenter und enthalten immer mehr Technik, um den Fahrer zu unterstützen und zu unterhalten (z.B. Navigation, Audio, Video, Telefon-/Datennetzwerk, Diagnose des Fahrzeugs, Fehlerauskunft, ...). Die Elektronik in einem modernen Fahrzeug besteht aus vielen kleinen 8 oder 16 Bit Mikroprozessoren, die miteinander kommunizieren. Dies verursacht große Kosten für die Verbindung der einzelnen Bauteile, schafft aber Unabhängigkeit und Sicherheit. Es wäre jedoch kosteneffizienter, die Funktionen in einer einzigen Einheit, bestehend aus ein paar rechenstarken Prozessoren, zu konzentrieren. Hierfür ist die Virtualisierungstechnologie geeignet, um unterschiedliche Systemtypen zu bedienen. Für jede Virtual Machine muss dabei die Verwendung von CPU, Memory und anderen Ressourcen präzise kontrolliert werden, da hierbei auch Safety-Critical Systeme verwaltet werden müssen. Trusted und Untrusted Software muss gehostet und voneinander streng abgeschirmt werden, um die nötige Sicherheit garantieren zu können (Nicht, dass z.B. durch einen Fehler im MPEG Stream des Infotainment-Systems plötzlich alle Airbags aktiviert werden).

Luftabwehr Computerbasierte Verteidigungssysteme in Luft- und Raumfahrt wurden früher einzeln angefertigt, um den strengen Anforderungen des Militärs, der zivilen Luftfahrt und der Weltraumforschung gerecht zu werden. Seit 1990 ist ein Umbruch im Gange. Sogenannte COTS (Commercial Off-The-Shelf) Systeme werden bevorzugt eingesetzt, um deren Vorteile auszunutzen. Dabei muss es vor allem möglich sein, dass Mission-Critical Software den vertragsgemäßen Auflagen und Verpflichtungen gerecht wird. Gast Betriebssysteme müssen im Fehlerfall schnell neu gestartet werden können, Zertifizierung muss möglich sein und die bestehende Software sollte schnell und einfach in das neue System migriert werden können.

Industrielle Automatisierung In der industriellen Automatisierung werden Echtzeitsysteme benötigt, die aber auch eine vertraute grafische Oberfläche bieten (wie z.B. CAD Workstations). In den letzten 10 Jahren wurden diese Anforderungen durch Microsoft Windows oder Sun Solaris basierte Systeme gelöst. Die Hardware entspricht dabei einem normalen PC, der mit einem zweiten System verbunden ist (meist durch eine PCI-Karte), das die Elektronik der Produktionsmaschine steuert. Dabei gibt es jedoch Probleme in den Bereichen Inter-System-Kommunikation, Integration, Test, Anwendungsentwicklung und Wartung. Diese Probleme würden nicht bestehen, wenn das Computersystem und die Komponente mit der Echzeitanforderung in einem System vereint werden. OKL4 kann dabei das grafische User-Interface bieten, aber gleichzeitig auch die Mission-Critical Software ausführen. Durch ver-

schiedene Virtual Machines kann so den Performance-Bedürfnissen der jeweiligen Anwendung gerecht werden.

7 Fazit

Mithilfe des seL4 Kernels ist es möglich, einen verifizierten fehlerfreien Systemkern zu schaffen. Durch die Verwendung von Capabilities im Kontext des erweiterten Take-Grant Access Model wird es möglich, einzelnen Applikationen fein granulare Rechte zu geben. So bietet das System die nötige Basis, um das Principle of Least Authority (POLA) einhalten zu können. Durch ein effizientes Memory Management, das es erlaubt, den Speicherverbrauch einer Applikation genauestens zu überwachen, ist es möglich, verschiedenen Attacken böswilliger Personen standzuhalten. Mithilfe einer sicheren und trotzdem schnellen Inter-Prozess Kommunikation und Isolation Domains können untrusted Applikationen ausgeführt werden ohne dass sie dem System Schaden zuführen oder Informationen erreichen, die nicht für sie bestimmt sind.

7.1 Aktueller Stand

Die Implementation des seL4 Kernels in Haskell ist beendet. Der Kernel Prototyp kann in einem Simulator ausgeführt werden, um die API weiterzuentwickeln. Es wurde außerdem bewiesen, dass die API isolations-fähig ist. Weiterhin wurde eine noch unreife Version in C programmiert, die nun weiter ausgebaut wird, um auch Performance-Tests durchführen zu können [16].

Literatur

[1] CAMBRIDGE, University of: *Automated Reasoning Group HOL page*. `http://www.cl.cam.ac.uk/research/hvg/HOL/`. Version: 2008. – [Online; Stand 18. August 2008]

[2] CAMBRIDGE, University of ; MÜNCHEN, Technische U.: *Isabelle*. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`. Version: 2008. – [Online; Stand 20. August 2008]

[3] CHEN, Hao ; DEAN, Drew ; WAGNER, David: Model Checking One Million Lines of C Code / UC Berkely. 2004. – Forschungsbericht

[4] COCK, David ; KLEIN, Gerwin ; SEWELL, Thomas: Secure Microkernels, State Monads and Scalable Refinement / National ICT Australia and School of Computer Science and Engineering, UNSW, Sydney, Australia. 2008. – Forschungsbericht

[5] DERRIN, Philip ; ELKADUWE, Dhammika ; ELPHINSTONE, Kevin: *seL4 Reference Manual*

[6] DERRIN, Philip G.: *A Secure Microkernel*, The University of New South Wales School of Computer Science and Engineering, Diplomarbeit, 2005

[7] DRESDEN, Technische U.: *DROPS - The Dresden Real-Time Operating System Project*. `http://os.inf.tu-dresden.de/drops/overview.html`. Version: 2008. – [Online; Stand 18. August 2008]

[8] ELKADUWE, Dhammika ; DERRIN, Philip ; ELPHINSTONE, Kevin: Kernel data First class citizens of the system / National ICT Australia and University of New South Wales Sydney, Australia. 2005. – Forschungsbericht

[9] ELKADUWE, Dhammika ; DERRIN, Philip ; ELPHINSTONE, Kevin: Kernel Design for Isolation and Assurance of Physical Memory / National ICT Australia and University of New South Wales Sydney, Australia. 2008. – Forschungsbericht

[10] ELKADUWE, Dhammika ; KLEIN, Gerwin ; ELPHINSTONE, Kevin: Verified Protection Model of the seL4 Microkernel / National ICT Australia and University of New South Wales Sydney, Australia. 2007. – Forschungsbericht

[11] ELPHINSTONE, Kevin ; HEISER, Gernot ; HUUCK, Ralf ; PETTERS, Stefan M. ; RUOCCO, Sergio: L4Cars / National ICT Australia and University of New South Wales Sydney, Australia. 2008. – Forschungsbericht

[12] ERÝ, Ondrej: *On Provably Correct Operating Systems*. 2007

[13] ETFORECASTS: *Worldwide Internet Users Top 1.2 Billion in 2006*. `http://www.etforecasts.com/pr/pr207.htm`. Version: 2008. – [Online; Stand 1. September 2008]

[14] HEISS, H.-U.: *Formale Modell der Zugriffskontrolle*. 2008

[15] HEISER, Gernot: Your System is Secure? Prove it! / NICTA and University of New South Wales and Open Kernel Labs Sydney, Australia. 2007. – Forschungsbericht

[16] HEISER, Gernot ; ELPHINSTONE, Kevin ; KUZ, Ihor ; KLEIN, Gerwin ; PETTERS, Stefan M.: Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level / NICTA and University of New South Wales. 2007. – Forschungsbericht

[17] KARLSRUHE, Universität: *L4Ka::Pistachio microkernel*. http://l4ka.org/projects/pistachio/. Version: 2008. – [Online; Stand 18. August 2008]

[18] L4HQ: *L4H! - L4 Kernel Projects*. http://l4hq.org/projects/kernel/. Version: 2008. – [Online; Stand 25. August 2008]

[19] LABS, Open K.: *Market Solutions*. http://www.ok-labs.com/markets/market-solutions. Version: 2008. – [Online; Stand 18. August 2008]

[20] MEDIA, Informa Telecoms ⋮ *Statistik: Weltweit 3,3 Milliarden Handy-Verträge*. http://www.inside-handy.de/news/10425.html. Version: 2007. – [Online; Stand 25. August 2008]

[21] NICTA: *Software: Project overview*. http://ertos.nicta.com.au/software/. Version: 2008. – [Online; Stand 25. August 2008]

[22] NICTA: *Technical Information seL4*. http://ertos.nicta.com.au/research/sel4/tech.pml. Version: 2008. – [Online; Stand 25. August 2008]

[23] NICTA ; UNSW: *The L4.verified project*. http://ertos.nicta.com.au/research/l4.verified/. Version: 2008. – [Online; Stand 15. August 2008]

[24] NICTA, ERTOS: *Probabilistic Temporal Analysis of Operating Systems Code (Potoroo)*. http://ertos.nicta.com.au/research/potoroo/. Version: 2008. – [Online; Stand 13. August 2008]

[25] OPERATING SYSTEMS GROUP DRESDEN, Technische Universität D.: *L^{4}Linux*. http://os.inf.tu-dresden.de/L4/LinuxOnL4/. Version: 2008. – [Online; Stand 20. August 2008]

[26] RINGERT, Jan O.: Eine Einführung in die funktionale Programmierung mit Haskell / Technische Universität Carolo-Wilhelmina zu Braunschweig. WS 07/08. – Forschungsbericht

[27] SCHWARZ, Benjamin ; WAGNER, Hao C. ; MORRISON, Geoff ; WEST, Jacob: Model Checking An Entire Linux Distribution for Security Violations / University of California, Berkeley. – Forschungsbericht

[28] TEAM FIASCO, Technische Universität D.: *FIASCO*. http://os.inf.tu-dresden.de/fiasco/. Version: 2008. – [Online; Stand 25. August 2008]

[29] WIKIPEDIA: *L4 microkernel family — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=L4_microkernel_family&oldid=232380031. Version: 2008. – [Online; Stand 12. August 2008]

[30] WIKIPEDIA: *Mach (kernel) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Mach_(kernel)&oldid=232520516. Version: 2008. – [Online; Stand 25. August 2008]

[31] WIKIPEDIA: *Trusted computing base — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Trusted_computing_base&oldid=226372348. Version: 2008. – [Online; Stand 1. September 2008]