SOK - Sichere Betriebssystemarchitekturen

Christopher Zell
zell.christopher@fu-berlin.de

***Zusammenfassung*—Sichere Systeme bestehen aus verschiedenen Schichten, zum einen aus User Interface, Dateisysteme, trusted computing base (TCB) z.B. Kernel aber auch grundlegende Sicherheitsmechanismen. Die richtige Kombination für ein sicheres System ist dabei sehr wichtig.**

Auf Kernel, Sicherheitsmechanismen sowie einige Kombinationen bzw. sichere Betriebssysteme soll in diesem Systematization of Knowledge (SOK) eingegangen werden. Es werden dabei Mirco-, Exokernel sowie Hybride Systeme und Zugriffsmechanismen wie Protection Rings, Access Bits und Capabilities vorgestellt und erläutert. Es wird versucht bei den einzelnen Konzepten Bezug auf die heutigen Systeme und die Verwendung von Mechanismen zu nehmen, dies soll vor allem zeigen welche Konzepte sich durchgesetzt haben.

I. Einleitung

Sichere Systeme bestehen aus verschiedenen Schichten zum einen aus User Interface, Dateisysteme, trusted computing base (TCB) z.B. Kernel aber auch grundlegende Sicherheitsmechanismen. Die richtige Kombination für ein sicheres System ist dabei sehr wichtig.

Auf Kernel, Dateisysteme, Sicherheitsmechanismen sowie einige Kombination bzw. sichere Betriebssysteme soll in diesem Systematization of Knowledge (SOK) eingegangen werden.

SOK wurde für die IEEE Symposium on Security and Privacy (SOSP) Konferenz entwickelt und soll die bereits existierenden Arbeiten und Quellen zusammenfassen und in einem Paper vereinigen, sodass ein schneller Blick auf bereits getane Arbeit ermöglicht wird. Denn das Problem was die Veranstalter der Konferenz erkannt haben ist, das viele Dinge die bereits in der Vergangenheit entwickelt oder erforscht wurden in Vergessenheit geraten [1].

Deshalb werden in diesem SOK einige historische Paper verwendet, deren Arbeit zusammengefasst und auf den Punkt gebracht. So werden z.B. Mikro-, Exokernel sowie andere Hybride Systeme aus den folgenden Papers [2]–[6] vorgestellt. Des Weiteren werden Zugriffsmechanismen wie Access Bits, Protection Rings und auch Capabilities erläutert und anhand von Beispiel Systemen aufgezeigt. Die folgenden Papers wurden dabei verwendet: [7]–[11].

A. Motivation

Seitdem die Idee der Entwicklung von Mehrbenutzersysteme existiert wurde auch darüber nachgedacht wie man die einzelnen Benutzer isolieren kann. Aber nicht nur über die Isolierung wurde nachgedacht sondern auch grundlegend wie man die Daten der jeweiligen Nutzer vor unberechtigten Zugriffen schützen kann. Die Entwicklung der Betriebssysteme hat gezeigt das es verschieden und gute Lösungen gibt um ein sicheres Betriebssystem zu entwickeln. Jedoch scheiterte es oft entweder an der Umsetzung, Performanz oder an der Nutzbarkeit. In dieser Arbeit werden die Ideen für die verschiedenen Sicherheitsmechanismen zusammengetragen, falls möglich wird versucht diese zu vergleichen und zu zeigen welche sich evtl. sogar durchgesetzt haben.

II. Sicherheitsmechanismen

In diesem Abschnitt werden zum einen Protection Rings (siehe Abschnitt II-A), Access Bits bzw. Access Matrix (siehe Abschnitt II-B) und Capability Systeme (siehe Abschnitt II-C) erklärt und auf die bereits erstellten Arbeiten verwiesen.

A. Protection Rings

In folgendem Abschnitt wird kurz auf das Multiplexed Information and Computing Service (Multics) Betriebssystem aus [7] eingegangen und dieses erläutert, danach werden mithilfe von Multics die Protection Rings erklärt.

1) Multics: In Multics werden Informationen in Segmente gespeichert und können geteilt werden, sie beinhalten eigene Attribute über deren Größe sowie die Zugriffsrechte. Segmentierung liefert eine allgemeine Basis für den direkten Zugriff und das teilen von online Informationen durch erfüllen zweier Design Ziele.

1) direkter Zugriff
2) Zugriffskontrolle

Vorteil der direkten Adressierung ist das die Informationen nicht mehr kopiert werden müssen, da alle Instruktionen und Daten im System vom Prozessor adressierbar sind. D.h. Duplizierung von Prozeduren und Daten ist unnötig, Abbilder von Prozessen müssen nicht vorher geladen werden die originalen Prozeduren können direkt adressiert werden. Dies soll dem Programmierer eine attraktive Reduktion der Programm-Komplexität versprechen.

Multics Segmente sind Pakete von Informationen welche direkt adressiert und auf die kontrolliert zugegriffen werden können. Mit jedem Segment sind Zugriffsattribute für jeden User, der auf dieses Segment evtl. zugreifen will, assoziiert. Diese Attribute werden von der Hardware überprüft. Jede Information kann direkt als Segment referenziert werden. In anderen Systemen, meist online Informationen, werden diese als Dateien referenziert. (Unix, Plan 9 etc.)

In den meisten Systemen, die Informationsteilung zur Verfügung stellen, sind die genannten beiden Kriterien nicht erfüllt. Als Beispiel wird im Paper [7] das CTSS System genannt, welches Informationen die geteilt werden sollen in Dateien vorhält. Das bedeutet das eine Kopie in den Buffer des Users kopiert werden muss damit der Benutzer darauf zugreifen kann. (Benötigt IO - Request)

In nicht segmentierten Systemen macht das benutzen von Kern Abbildern es so gut wie unmöglich die Zugriffskontrolle durchzusetzen.

Die Hardware hat auf jedes Segment mithilfe eines Segmentdeskriptors, der die Attribute beinhaltet, Zugriff. Name und Attribute eines Segments sind mit einem Katalog assoziiert. Im Grunde existiert eine Tabelle mit je einem Eintrag für ein Segment. Ein Eintrag beinhaltet den Namen sowie die Segmentattribute. In Multics ist dieser Katalog als mehrere Segmente implementiert und wird Verzeichnis bzw. Directory genannt. Das Verzeichnis wird in einem Baum strukturiert. D.h. ein Segmentname besteht aus einer Liste von einzelnen Subnamen die die Position des Eintrags im Baum reflektieren. (Subname = Entryname, Liste = Pfadname) Der Pfadname ist in der ganzen Hierarchie einzigartig und der Entryname nur im gegebenen Directory.

Multics gibt dem Nutzer eine große Menge an Segmentdeskriptoren, sodass das Zwischenspeichern der Informationen nicht mehr notwendig ist und die Informationen nicht ihre Zugehörigkeit verliert. Zudem heißt das, dass ein Multics Nutzer keine Dateien benutzt sondern alle Informationen als Segmente referenziert, welche direkt vom Programm erreichbar sind.

Für Multics Nutzer erscheint der Speicher als eingeschlossener großer unabhängiger linearer Kernspeicher, jeder assoziiert mit einem eigenen Deskriptor. Der Zugriff erfolgt durch [*Name*, *i*] wobei *Name* der symbolische Name des Segments und *i* das Wort im linearen Speicher ist. Jeder Nutzer kann auf [*Name*, *i*] referenzieren, jedoch hat er nur die Zugriffsrechte die er auch vom Ersteller des Segments zugewiesen bekommen hat und die im Segmentdeskriptor eingetragen sind. Kombinationen von den folgenden Zugriffsrechten ”read”, ”write”, ”execute” und ”append” sind möglich.

Segmente können zudem in gleich große Teile geteilt werden, sogenannten Pages. Dieser Ansatz der Segmentierung und des Paging wird auch in heutigen Systemen noch verwendet.

Im Paper [7] wird auf die Vorteile hingewiesen das die Allokierung durch das Paging sehr vereinfacht wird, sowie das nur die Pages, welche auch benötigt werden, im Speicher sein müssen. Das bedeutet das Segmentgrössen keine Einschränkungen mehr besitzen.

Sie weisen auch darauf hin das die Pages, die gebraucht werden, angefordert werden können und das keine anderen in den Speicher mit übertragen werden müssen.

Dies wird auch heutzutage in aktuellen Betriebssystemen noch betrieben, sogenanntes Paging welches mithilfe von verschieden Algorithmen wie FIFO (First In First Out), LRU (Last Recently used) oder anderen umgesetzt wird.

Was heutzutage der Translation Lookaside Buffer (TLB) bzw. die Memory Managment Unit (MMU) macht (Seitennummern in Speicheradressen um mappen) hat in Multics der Supervisor gemacht. Dieser transformierte den symbolischen Namen in eine passende Hardware Adresse, welche direkt vom Prozessor für weitere Referenzierungen benutzt wird.

In der Multics Implementierung sind alle Segmente in Pages unterteilt und die Page size beträgt 1024 Wörter. Prozesse sind Programme in der Ausführung (vgl. [7][S. 34]). Jeder Prozess in Multics besitzt ein privates Deskriptor Segment, welches Segmentnummern in Kernspeicheradressen umwandelt und eine Private Tabelle besitzt die die symbolischen Segmentnamen in Segmentnummern abbildet. Diese Tabelle wird auch als Known Segment Table (KST) bezeichnet. Der Supervisor in Multics ist im Grunde wie der Kernel in heutigen Unix bzw. Linux Systemen.

Nicht wie in anderen Systemen arbeitet er im eigenem Speicherraum oder Prozess, sondern die Supervisor Prozedur und deren Daten werden mit jedem Prozess geteilt. Immer wenn ein neuer Prozess erstellt wird, wird das Deskriptor Segment mit den Supervisordeskriptoren initialisiert. Dadurch müssen aber auch die Supervisor Segmente vor unautorisierten Zugriffen geschützt werden. Multics unterstützt den Benutzer mit einem sogenannten Ring Protection Mechanismus, welches die Segmente in dem Adressraum in einzelne Mengen mit verschiedenen Zugriffsrechten unterteilt. Dieser Mechanismus wird vom Supervisor zum eigenen Schutz verwendet. Siehe dazu nächster Abschnitt II-A2

2) Protection Rings: In folgendem Abschnitt werden die bereits erwähnten Protection Rings anhand von Multics näher erläutert. Multics wurde zuerst auf einem Honeywell 645 Computersystem ausgeführt, jedoch konnte die Hardware nicht viele Zugriffsmechanismen zur Verfügung stellen. Deshalb wurde eine neue Hardware entwickelt die die Zugriffsmechanismen mehr auf Hardwareebene bringen sollte, sodass das meiste nicht mehr über Software geprüft werden muss. Das heißt die sogenannten Protection rings wurden so gut wie vollständig in Hardware umgesetzt. [8]

Schutz soll so gewährleistet werden dass einzelne Be-

nutzer von einander getrennt sind und wenn gewünscht aber untereinander Kommunizieren können. Vier Kriterien können auf Zugriffskontrollmechanismen angewendet werden:

1) *Funktionale Capability*
2) *Economy*
3) *Simplicity*
4) *Programming generality*

Das erste bedeutet es soll bestimmte Benutzerschutzbedürfnisse tilgen, *economy* befasst sich dabei mit den Kosten der Umsetzung. Kosten beinhaltet dabei auch Komplexität des Subsystems, Benutzer Belästigung durch benutzen dieser Zugriffsmechanismen sowie der benutzte extra Speicher und die erhöhte Laufzeit. Die Kosten sollten Proportional zu den Leistungen sein. *Simplicity* bedeutet Einfachheit, das heißt das die Zugriffsmechanismen so einfach wie möglich sein sollen. Oft impliziert das fehlen von Einfachheit auch das fehlen von Sicherheit. *Programming generality* besagt einfach das die Prozeduren so allgemein gehalten sind das sie ohne Wissen der internen Struktur verwendet und Dinge auch ersetzt werden können. Es ist schwer ein Zugriffskontrollmechanismus zu entwickeln mit der Erfüllung aller Kriterien.

Eine Domäne ist eine Menge von Zugriffsberechtigungen. Ein sehr allgemeiner Zugriffskontrollmechanismus würde kein Beschränkung von der Anzahl an Domänen für ein Prozess machen und würde keine einschränkenden Beziehungen für die Zugriffsberechtigungen aufzwingen. Unglücklicherweise ist das Entwickeln eines solchen Mechanismus der auch die letzten drei Kriterien *economy*, *simplicity* und *programming generality* erfüllt ein schwieriges Forschungsthema. In Multics wurden die Anzahl von Domänen für ein Prozess limitiert und es werden Beziehungen von den Zugriffsrechten innerhalb der Domänen aufgezwungen. Das Resultat sind sogenannte Protection rings. (Siehe [8, S. 160])

Die Charakterisierung von Ringen als beschränkte Implementierung von Domänen ist das Ergebnis späterer Einsicht. Als sie Entwickelt wurden, wurden Ringe als natürliche Verallgemeinerung der Supervisor/User Modi, die Schutz in vielen Computern liefern, betrachtet.

Mit jedem Prozess sind eine bestimmte Anzahl von Domänen assoziiert, genannt Protection Rings. Diese r Ringe sind mit Zahlen von 0 bis $r-1$ benannt. Die Zugriffsberechtigungen in Ring m sind so beschränkt das sie eine Teilmenge von n sind, wenn gilt: $m > n$. Das bedeutet Ring 0 besitzt die größte Menge an Berechtigungen und $r-1$ die kleinste. Also hat ein Prozess der im Ring 0 ausgeführt wird die meisten Zugriffsprivilegien und im Ring $r-1$ die wenigsten. Es existieren sogenannte brackets (engl. Klammern) für die einzelnen Berechtigungen wie *read*, *write* und *execute*. D.h. ein write bracket gibt an in welchem Bereich das write Recht angewendet werden darf. Neben den Berechtigungen werden noch die Ausmaße und ein Flag für die jeweilige Berechtigung gespeichert. Ist das Flag auf *on* gesetzt so wird die obere Grenze des Bereichs für die Berechtigung gespeichert, wobei der Ring r kleiner oder gleich der Grenze sein muss. Ist das Flag auf *off* gesetzt so gibt es keine Berechtigung in keinem Ring für diesem Prozess. In Multics wurden 8 Ringe als angemessen angenommen und umgesetzt, jedoch ist das keine Vorgabe und in anderen Systemen können andere Anzahlen vorkommen.

Eine neue Berechtigung wurde eingeführt und zwar das ändern der Ausführungsdomän eines Prozesses. Durch das ändern einer Domäne können neue Berechtigungen dazu kommen, weshalb es eine Operation ist die gut Kontrolliert werden muss. Mithilfe der Domänen soll es möglich sein das bestimmte Zugriffsberechtigungen nur an bestimmten stellen Verfügbar sind bzw. wenn bestimmte Programmteile ausgeführt werden. Das heißt an bestimmten Programmpositionen auch *gates* genannt soll es möglich sein die Domäne zu ändern. Es wurden dafür *gate list* und *gate extension* eingeführt die eine neue bracket definieren in der das Programm die Ringe wechseln kann, diese gelten nur für das herunter wechseln der Ringe, da dort mehr Rechte verfügbar werden. Wechselt man ein Ring nach oben so werden es weniger Rechte deshalb soll das ohne Beschränkung möglich sein.

In Multics wird der Supervisor und die Protection Rings so umgesetzt das die Primitiven Operationen wie Zugriffskontrolle, I/O, Speicherverwaltung sowie Prozessverwaltung im Ring 0 ausgeführt werden, alles andere vom Supervisor im Ring 1.

Die Hardware Umsetzung der Protection Rings, die im Paper [8] beschrieben wurde, ermöglicht die drei folgenden Dinge:

- Benutzer können beliebige geschützte Subsysteme erstellen, die von anderen genutzt werden können
- Der Supervisor kann in Schichten Implementiert werden, die aufgezwungen werden.
- Der Nutzer kann sich selbst schützen während er eigene oder andere Programme debugged.

B. Access Bits oder Matrix

In folgendem Abschnitt werden die Access Bits bzw. Access Matrizen als Zugriffsmechanismus anhand der Betriebssysteme Unix und Plan 9 beschrieben.

1) Unix: Unix wurde von den bekannten Informatikern Dennis Ritchie (C) und Ken Thompson (Unix, UTF8, Plan9 etc.) an den Bell Labs entwickelt. Es ist ein

Universal-, Mehrbenutzer- und interaktives Betriebssystem. Ein großer Teil von Unix ist in C geschrieben, was vor allem Multiprogramming und das Teilen von Code ermöglicht. Die wichtigste Aufgabe von UNIX ist ein Dateisystem bereitzustellen. Es gibt drei verschiedene Arten von Dateien (o. Files): gewöhnliche Datenträgerdateien, Verzeichnisse, und spezielle Dateien. Anders als in Multics ist das ganze System in Dateien aufgeteilt und nicht in Segmente. Gewöhnliche Datenträgerdateien beinhalten die Information die der User in diese platziert. Die Struktur wird nicht vom System kontrolliert, maximal von den zu benutzenden Programmen. Verzeichnisse liefern das Mapping zwischen den Dateinamen und den Dateien selber, somit entsteht eine Baumstruktur. Ein Verzeichnis ist im Grunde wie eine normale Datei nur das sie nicht von unprivilegierten Nutzern beschrieben werden kann. D.h. das System kontrolliert den Inhalt der Verzeichnisse. Das System besitzt eigene Verzeichnisse und Unterverzeichnisse, ein wichtiges ist *root*. Alle Dateien im System können, durch folgen eines Pfades von einer Kette von Verzeichnissen, gefunden werden. Der Startpunkt ist oft dabei *root*, vgl. [9, S. 366].

Das Zugriffskontrollschema ist in UNIX relativ simpel. Jeder Nutzer besitzt eine eigene ID, wird eine Datei vom Nutzer erstellt erhält diese die ID, sodass der Ersteller immer erfasst werden kann. Zudem gibt es 7 Bits, für die Dateien, die den Zugriff kontrollieren. Die ersten drei geben für den Eigentümer die Rechte: *read*, *write* und *execute* an. Die nächsten drei für alle anderen Nutzer die Rechte und das letzte Bit ist das sogenannte S-Bit. Ist dieses Bit gesetzt so wird die User ID temporär mit der des Datei Eigentümers ersetzt bzw. das Programm mit der User ID des Eigentümers ausgeführt. D.h. es werden deren Rechte benutzt für den Zeitraum der Programmausführung. Dies sollte ermöglichen das Programme auf Dateien zugreifen können für die der Nutzer im Normalfall keine Rechte besitzt. (Siehe [9, S. 367] Mithilfe des S-Bits konnten jedoch einige Exploits geschrieben werden die es unautorisierten Nutzern ermöglichten die Kontrolle über das System zu erlangen.

UNIX wurde laut den Autoren auch von Multics beeinflusst vor allem in dem Bereich I/O, Shell und deren Funktionalität. (Siehe [9, S. 374]) UNIX hat sich bis heute stark durchgesetzt bzw. basieren viele heutige Systeme auf dem UNIX System.

2) Plan 9: Laut den Autoren von Plan 9, hatte UNIX einige Probleme die zu tief lagen und somit nicht einfach gelöst werden konnten. Es gab jedoch auch einige Ideen die in dem neuen Bell Labs System übernommen werden konnten. Das war zum einem das Dateisystem um Ressourcen erreichbar zu machen und zu benennen.

Die Sicht auf das System ist auf drei Prinzipien gebaut. Erstens die Ressourcen werden benannt und sind erreichbar genauso wie Dateien in einem Hierarchischem Dateisystem. Zweitens es existiert ein Standard Protokoll namens *9P*, um auf die Ressourcen zuzugreifen. Drittens die unterschiedlichen Hierarchien die von den verschiedenen Services geliefert werden sind verbunden zu einem einzelnen Privaten Hierarchischem Dateinamensraum. D.h. es existieren sogenannte lokale Namensräume auf den nur die jeweiligen Nutzer Zugriff haben. Dieses Feature wird vor allem durch den eigenen Fensterserver benutzt der aus bestimmten Dateien im lokalen Namensraum Daten liest die er dann dem Nutzer anzeigt.

9P ist ein Netzwerkprotokoll welches Maschinenzugriff auf entfernten Dateien ermöglicht. Es ist als Menge von Transaktionen strukturiert das Anfragen von einem Client zu einem Server und die Antwort zurück sendet. 9P kontrolliert das Dateisystem nicht nur die Dateien. Es beinhaltet Prozeduren um Dateinamen aufzulösen und um die Namenshierarchie vom Dateisystem eines Servers zu traversieren. Dateizugriff ist auf Byte und nicht auf Blockebene, nicht wie bei *NFS* und *RFS*.

Eine große Plan 9 Installation besitzt eine Anzahl von Computer die zusammen in einem Netzwerk verbunden sind, jeder liefert eine eigene Klasse von Anwendung. Verteilte Multiprozessorserver stellen Berechnungszyklen bereit, andere große Maschinen stellen Speicher zur Verfügung. Es existieren sogenannte Workstations welche gleichzusetzen sind mit den Terminals der alten *time sharing machines*. Wenn jemand das System durch ein Terminal nutzt ist es durch den Benutzer personalisiert.

Das Dateisystemmodel ist gut verstanden bei Systementwicklern aber auch bei allgemeinen Nutzern. D.h. Services die Datei ähnliches Interface bereitstellen sind leicht zu bauen, leicht zu verstehen und leicht zu benutzen. Das ist auch wirklich eines der wenigen Betriebssystem die mit an die Nutzbarkeit gedacht haben um es so einfach wie möglich für die Nutzer zu machen.

Der Dateiserver hat drei Speicherlevel, erstens den *Memory Buffer*, zweitens die *Disk* und drittens den *Massenspeicher* in einem *write-once-read-many* (WORM) Gerät. Die *Disk* ist dabei der Cache für den WORM und der *Memory Buffer* ist der Cache der *Disk*. Ein weiteres Feature existiert, welches es ermöglicht zu bestimmten Zeitpunkten einen *dump* vom Dateisystem zu erstellen. Das Dateisystem wird dabei eingefroren und alle modifizierten Blöcke seitdem letztem dump werden in ein Warteschlange geschrieben, die dann nach und nach auf dem WORM geschrieben werden. Auf die *dumps* hat man direkten Zugriff, sodass man z.B. alte Compiler Versionen und vieles weitere testen kann. Zudem kann auch ein komplettes Systembackup erfolgen was aber

einige Tage dauert, da der Cache vom WORM erst wieder eingelesen werden muss etc. Die Zugriffsrechte der *dump* Files sind die selben wie zuvor, jedoch ist das Dateisystem nur *read-only*. D.h. Dateien können nicht im *dump* geschrieben werden, egal was für Zugriffsbits gesetzt sind.

Die einheitliche Verbindung von Komponenten in Plan 9 macht es möglich die Installation in vielen Wegen zu Konfigurieren.

Authentifizierung wird für den Benutzerzugriff auf eine Ressource verwendet. Der Nutzer der die Ressource Anfragt wird Client genannt und der Nutzer der den Zugriff gewährt wird Server genannt. Jeder Plan 9 User hat ein assoziierten DES Authentifikationsschlüssel. Die Identität des Nutzers wird verifiziert durch die Möglichkeit bestimmte Nachrichten *challenges* zu ver- und entschlüsseln.

Die Authentifizierung wird wie folgt durchgeführt: Nachdem die *challenges* ausgetauscht wurden, Kontaktiert eine Partei den Authentifizierungsserver um verschlüsselte Berechtigungstickets, mit dem Schlüssel jeder Partei, zu erstellen. Dieses Ticket beinhaltet ein neuen Konversationsschlüsseln. Jede Partei entschlüsselt sein eigenes Ticket und benutzt den Konversationsschlüssel um die *challenge* der anderen Partei zu entschlüsseln. Plan 9's Authentifikation ermöglicht eine sogenannte *Spricht für* Beziehung, sodass ein Benutzer die Befugnis eines anderen besitzt.

Plan 9 besitzt keine Superuser sowie sie in UNIX existieren. Jeder Server ist verantwortlich für das aufrechterhalten der eigenen Sicherheit. Normalerweise wird der Zugriff nur über die Konsole, welche mit einem Passwort geschützt ist, erlaubt. Wenn eine Datei durch ein Nutzer *read* geschützt ist kann nur dieser Nutzer Zugriff an andere weiter geben. Es existiert ein sogenannter administrativer Nutzer *adm* der besondere Rechte besitzt. None ist zudem ein Benutzer der immer existiert und der kein Passwort besitzt. Diesem ist es immer erlaubt sich zu verbinden. None besitzt nur eingeschränkte Nutzungsrechte. Die Idee hinter diesem Nutzer ist die selbe wie hinter dem anonymen Nutzer in FTP.

Ähnlich wie in UNIX besitzen die Dateien sogenannte Zugriffsbits, welche die *read*, *write* und *execute* Rechte anzeigen. Zum einen für den Besitzer der Datei, für eine bestimmte Gruppe und für alle restlichen. Die Gruppen sind ein neues Feature in Plan 9 welches auch heute noch in Linux verwendet wird. Gruppen sind nichts anderes als ein Nutzer mit einer Liste von anderen Nutzern in dieser Gruppe. Für Gruppen können sogenannte *group leader* existieren, diese können die besagte Gruppe dann Verwalten. Normalerweise gehört eine Datei zu dem Nutzer der diese erstellt hat. Der Gruppenname wird von dem Verzeichnis in dem die neue Datei liegt vererbt. In Linux ist es ähnlich, nur der Unterschied liegt darin das eine Datei die default Gruppe des Nutzers erbt und nur *root* eine Gruppe verändern darf [12] [13].

C. Capabilities

In folgendem Abschnitt werden Capability Zugriffsmechanismen anhand vom Cambridge CAP Computersystem erklärt.

1) CAP: CAP wurde entwickelt um detaillierten Speicherschutz zu gewährleisten. Die Intention ist das jedes Modul eines Programms, welches auf dem CAP ausgeführt wird, nur Zugriff auf genau die und nur die Daten besitzen soll, welche sie auch wirklich für korrekte Funktionalität benötigt.

Capabilities werden benutzt um in CAP den Zugang zu Segmenten zu gewährleisten. Capabilities besitzen *base* und *limit* des Segments, sowie sogenannten *access status*. Dieser Status besteht aus 5 Bits, drei davon sind für den Datenzugriff also *read*, *write* und *execute*. Die anderen beiden sind für den Capability Zugriff, d.h.*read Capability* und *write Capability*. Es wird zwischen *data typ* und *Capability type* Capabilities unterschieden. Der erste Typ besitzt gesetzte Bits in den ersten drei Bits und der zweite Typ in den letzten. Es existiert keine Capability mit gesetzten *data* und *Capability access bits*. Capabilities werden in Segmenten gespeichert die *Capability type access* besitzen. Zu jeder Lebenszeit eines Prozesses sind nur die Ressourcen erreichbar die von den Capabilities definiert wurden. Mit jedem Prozess ist ein fundamentales Segment assoziiert, genannt Process Resource List (PRL). Im CAP Betriebssystem existiert eine Prozesshierarchie. Als Wurzel der Hierarchie steht ein einzelner Prozess deren PRL wird auch Master Ressource List genannt. Das Programm welches in diesem Prozess läuft ist Verantwortlich für die Erstellung und Verwaltung von Level 2 Prozessen es wird Master Coordinator genannt (MC). Die PRL eines Level 2 Prozesses ist ein normales Datensegment vom MC. Es ist möglich die Hierarchie zur erweitern und weitere Junior bzw. Subprozesse zu erzeugen. Diese benötigen nur eine Capability für ein Segment um dies als PRL zu nutzen.

In heutigen Linux Systemen werden auch sogenannte Prozesshierarchien verwendet, siehe z.B. die Shell die bei jedem Kommando, bis auf built-in Funktionen, ein neuen Kind-Prozess erzeugt und dieses Kind das Kommando ausführen lässt. Weiteres Beispiel ist das nachdem Boot ein sogenannter init Prozess erzeugt wird, der als Wurzel der Prozesshierarchie hierbei steht, ähnlich wie im CAP System [14].

III. Weitere sichere Systeme

Unabhängig von den zuvor bereits erwähnten Zugriffsmechanismen wollten einige Systementwickler und Wissenschaftler neue weitere sichere Systeme vor allem minimalistische Kernel entwickeln. Minimalistische Kernel um somit die Sicherheit eines Systems besser zu gewährleisten. Im folgendem Abschnitt wird dabei auf die Entwicklung des Mikro-, Exokernels und auf einige andere Versuche eingegangen sichere Systeme zu erstellen.

A. Mikrokernel

Grundidee des Mikrokernels ist das der Kernel minimiert und das alles so gut es geht ausgelagert werden soll. Die Vorteile dieser Herangehensweise ist das ein klares Mikrokernel Interface die modularität der Systemstruktur erhöht. Das System ist flexibler und anpassbarer, verschiedene Strategien und Schnittstellen können gleichzeitig im System existieren.

Dieser Abschnitt bezieht sich auf das Paper [2] welches die Erstellung eines Mikrokernel behandelt. Memory-Management sowie Paging werden im Mikrokernel außerhalb platziert, nur *grant*, *map* und *flash* Operationen sind im Kernel enthalten. [2, S. 238 f] Threads, IPC sowie das generieren von UID's (unique identifiers) werden jedoch im Kernel behalten.

Ein Mikrokernel kann höhere Schichten mit einem minimalen Satz von passenden Abstraktionen liefern. Diese sind flexible genug um beliebige Implementierungen von Betriebssystemen sowie das weite ausnutzen der Hardware zu erlauben. Multi-Level Sicherheitssysteme benutzen dazu noch sogenannte *clans* oder ein ähnliches Referenzmonitor Konzept. Liedtke zeigt das es mit seinem L4 Mikrokernel möglich ist einen performanten Mikrokernel durch prozessorspezifische Implementierung, von Prozessor unabhängigen Abstraktionen, zu erzeugen. [2, S. 248]

B. Exokernel

Betriebssysteme definieren die Schnittstelle zwischen den Anwendungen und den physikalischen Ressourcen. Unglücklicherweise kann diese Schnittstelle die Performanz sowie die Implementierung von Anwendungen einschränken. Üblicherweise abstrahieren Betriebssysteme Informationen über die Hardware in Prozessen, Dateien, Adressräumen und IPC. Diese Abstraktionen definieren eine virtuelle Maschine in der die Anwendungen ausgeführt werden. Deren Implementierung kann **nicht** durch nicht vertrauenswürdige Anwendungen ersetzt werden. Das führt zu einigen Problemen: Es verweigert Anwendungen den Vorteil von Domänen spezifischen Optimierungen. Es schränkt zudem die Flexibilität von Anwendungsentwicklern ein, da neue Abstraktionen nur durch sonderbare Emulationen zum System hinzugefügt werden können.

Diese Probleme wollten die Entwickler des Exokernels bekämpfen, deren Ziel es war ein minimalistischen Kernel genannt Exokernel zu erstellen. Dieser Kernel vielfälltigt sicher die verfügbaren Ressourcen. Der Rest des Systems besteht aus nicht vertrauenswürdigen Anwendungen, die z.B. IPC oder Ressource Management umsetzen. D.h. sogenannte *library operating systems* arbeiten über der Exokernel Schnittstelle. Diese Implementieren die Schnittstelle des höheren Levels. Das ermöglicht Anwendungen bestimmte Bibliotheken auszuwählen bzw. deren Implementierung direkt auf deren Notwendigkeiten anzupassen. Z.B. kann es verschiedene Arten von page-table Implementierungen in einem System geben. Eine Anwendung kann dann eine Bibliothek mit einer bestimmten Implementierung, welche am besten passt, auswählen.

Um Anwendungen die Kontrolle über die Hardware zu ermöglichen definiert der Exokernel ein Low-Level Interface. Der Exokernel exportiert Hardware Ressourcen, was eine effiziente und einfache Implementierung ermöglicht. Mithilfe von drei Techniken wird das Exportieren sicher umgesetzt. Zum einen mit sogenannten *secure bindings*, mit *visible resource revocation* und durch ein *abort protocol*.

Secure binding ist ein Schutzmechanismus der die Authentifizierung und die tatsächliche Benutzung von einander trennt. Die Überprüfungen sind entweder in einfachen Operationen im Kernel oder Hardware ausgedrückt, die leicht implementiert werden können. Die Authentifizierung findet nur zur Bindezeit statt, das ermöglicht das Verwalten vom Schutzmechanismus los zu lösen. Es gibt drei Basis Techniken um *secure bindings* umzusetzen. Zum einen mit Hardware Mechanismen, Software Caching sowie mit dem Downloaden von Anwendungscode. Downloading des Codes kann zum erhöhen der Performanz benutzt werden, es besitzt zwei Vorteile. Zum einen Eliminierung von Kernel Übergang (Switch) und die Ausführungszeit kann eingegrenzt werden. Der wesentliche Knackpunkt ist dieser das der Code ausgeführt werden kann wenn die Anwendung nicht scheduled ist. Diese Loslösung erlaubt die Ausführung von heruntergeladenen Code in Situationen an denen ein Kontextwechsel zur Anwendung nicht möglich ist.

Visible resource revocation soll es ermöglichen das Ressourcen die gebunden wurden wieder von einer Anwendung befreit werden bzw. deren *secure binding* brechen. Normalerweise setzen Betriebssysteme diese *revocation* invisible durch. D.h. sie allozieren oder deal-

lozieren Speicher oder Ressourcen ohne die Anwendungen zu informieren. Diese sichtbare *resource revocation* ermöglicht das *library operating systems* darauf reagieren können und z.B. den benötigten Prozessorzustand speichern. Jedoch einige Dinge wie Prozessorkontextbezeichner werden sehr oft zurückgeholt wodurch es besser ist die nicht sichtbare *resource revocation* zu nutzen.

Das *abort protocol* ermöglicht es, indem Fall das die Anwendung auf die Anfrage der *revocation* nicht zufriedenstellend reagiert hat, die Ressource zurück zu holen. Anstatt die Anwendungen, die nicht in bestimmter Zeit antworten einfach zu beenden werden die *secure bindings* aufgelöst und die *library operating systems* bzw. Anwendungen informiert.

Die Referenz Implementierung Aegis (Exokernel) mit dem ExOS (library operting system) gibt den Anwendungen größere flexiblität und Performanz als monolithische oder Mikrokernel Systeme. Jedoch ist der Exokernel Ansatz bis heute noch in der Forschung und wird noch nicht produktiv wie z.B. der Mikrokernel eingesetzt.

C. Spin

Spin ist ein Betriebssystem welches dynamisch spezialisiert werden kann, um sicher die Bedürfnisse von Performanz und Funktioniltät für Anwendungen anzupassen. Ähnlich dem Exokernel versucht Spin es zu ermöglichen das die Betriebssystemschnittstelle angepasst werden kann, sodass die Bedingungen für Anwendungen verbessert werden können. Soll heißen das bei normalen Betriebssystemen Anwendungen die nur schlecht oder gar nicht laufen in Spin die Schnittstellen des Betriebssystem auf die Notwendigkeiten angepasst werden können.

Spin und seine Erweiterungen sind in der typsicheren Sprache Modula-3 geschrieben. Modula-3 besitzt folgende Kernmerkmale: Interfaces, Typsicherheit, automatisches Speichermanagement, Objekte, generische Interfaces, Threads und Exceptions. Durch die Typsicherheit ist, lauten den Autoren von [4], gewährleistet das der Code nicht auf beliebigen Speicher zugreift, Pointer nur auf die Objekte des Zeigertyps verweisen etc. Das Spin Betriebssystem beruht auf vier Techniken die auf Sprach- oder Laufzeitebene implementiert sind. Die erste ist die *co-location*, d.h. die Erweiterungen werden dynamisch in den virtuellen Adressraum des Kernel verlinkt. *Co-location* ermöglicht Kommunikation zwischen System und Erweiterungscode mit niedrigen Kosten.

Zweitens *enforced modularity*, wie bereits erwähnt sind Erweiterungen in Modula-3 geschrieben welche eine modulare Programmiersprache ist und für die der Compiler Schnittstellengrenzen zwischen Modulen erzwingt. Erweiterungen die im virtuellen Adressraum des Kernels ausgeführt werden können auf keinen Speicher oder privilegierte Instruktionen zugreifen sofern sie nicht die explizite Erlaubnis durch ein Interface erhalten haben. Modularität erzwungen vom Compiler ermöglicht es Module mit geringen Kosten zu isolieren.

Die dritte Technik wird *logical protection domains* genannt. Erweiterungen existieren in logischen *protection domains* welche Kernel Namensräume sind, die Code und exportierte Interfaces beinhalten. Interfaces represäntieren Sichten auf die Systemressourcen die vom Betriebssystem geschützt sind. Ein im Kernel liegend dynamischer Linker löst Code in separaten logischen *protection domains* zur Laufzeit auf. Dies ermöglicht *cross-domain* Kommunikation mit dem Overhead eines Prozeduraufrufs.

Als letztes ist das *dynamic call binding* zu nennen. Erweiterungen werden als Antwort auf Systemevents ausgeführt. Ein Event kann jede potentielle Aktion im System beschreiben, wie z.B. ein page fault. Events werden in Schnittstellen deklariert und können abgefertigt werden mit dem Overhead eines Prozeduraufrufs, vgl. [4, S. 267 f].

Das sogenannte *protection model* kontrolliert die Menge an Operationen die auf die Ressourcen angewendet werden können. Alle Kernel Ressourcen in Spin sind mit Capabilities referenziert. Eine Capability in Spin ist eine unfälschbare Referenz auf eine Ressource. Spin implementiert die Capabilities direkt mithilfe von Pointern, nicht wie andere Betriebssysteme die dazu Hardware oder anderes benutzen. Siehe als Beispiel das CAP System welches Software und Hardware nutzt, vgl. [11]. Ein Pointer bzw. Zeiger kann vom Kernel zur User-Level Anwendung als veräusserte Referenz weitergegeben werden. Eine veräusserte Referenz ist ein Index, in der Anwendungstabelle, die typsichere Referenzen zur Kernel Datenstrukturen besitzt.

Die Erweiterungen verändern den Weg wie ein System ein Service zur Verfügung stellt. Jede Software ist irgendwie erweiterbar, aber das sogenannte *extension model* entscheidet wie leicht, transparent und effizient eine Erweiterung angewendet werden kann. Spin's *extension model* stellt eine kontrollierte Kommunikationsmöglichkeit zwischen den Erweiterungen und dem Basis System, welches verschiedene Interaktionsarten erlaubt, zur Verfügung. Erweiterungen in Spin sind in Events und Handler definiert. Ein Event ist eine Nachricht die eine Veränderung eines Zustands im System ankündigt oder eine Anfrage für einen Service. Ein Event-Handler ist eine Prozedur die dieses Event erhält und verarbeitet. Eine Erweiterung installiert einen Handler für ein Event durch explizites Registrieren, mithilfe eines Zentralen *dispatchers* vgl. [4, S. 272])

Das Spin Betriebssystem ermöglicht gute Performanz in einem erweiterbaren System, ohne die Sicherheit zu kompromittieren. Jedoch ähnlich wie der Exokernel ist Spin nur ein weiteres Forschungsthema und leider nicht Produktiv im Einsatz. Die Entwicklung von Spin wurde zudem nicht weiter verfolgt, was der Spin Homepage zu entnehmen ist. Dort werden seit 1996 keine weiteren Paper referenziert [15], zudem kann das System nur mit einer Linux Red Hat Version 4.2 gebaut werden welche im Mai 1997 erschien [16].

D. EROS

Eros ist ein Capability basierender Mikrokernel mit einem einstufigen Speichermodell. Die einstufige Speicher Persistierung ist transparent für Anwendungen. Speicherallokierung, Scheduling und fault Handling Politiken liegen außerhalb des Kernels, um verschiedene Operationsumgebungen und Anwendungsspezifische Ressourcemanagments zu erlauben. Eine Capability ist wie bereits schon erwähnt ist eine unfälschbares Paar von Objektidentifizierer und einer Menge von erlaubten Operationen (einem Interface) an diesem Objekt. Siehe dazu auch Abschnitt II-C oder III-C. Lauten den Autoren ist auch ein UNIX File Deskriptor eine Capability. In einem Capability System hält jeder Prozess Capabilities und kann diese Operationen ausführen welche durch die Capabilities autorisiert sind. Die Sicherheit wird dadurch gewährleistet das Capabilities fälschungssicher sind und Prozesse nur Capabilities durch autorisierte Schnittstellen erhalten. D.h es werden Capabilities nur an Prozesse gegeben die auch autorisiert sind diese zu halten.

Eine *protection domain* wie bereits erwähnt ist eine Menge von Capabilities, welche von einem Subsystem zugänglich sind. Die Idee eines Capability Systems ist die Unterteilung von Anwendungen und Betriebssystem in separate Komponenten, d.h. jede besitzt eigene *protection domains*.

EROS erstellt periodisch ständig Checkpoints vom System, um ein Konsistentes System sicherzustellen. Bei einem System mit 256 MB physikalischem Speicher dauert der Snapshot um die 50 ms. Was bei heutigen Systemen um einiges länger dauern würde. Die EROS Architektur ist in Kernel und Kernel Services eingeteilt, zudem beinhaltet die Architektur eine Menge von Systemservices die von nicht privilegierten Anwendungen implementiert sind. Der Kernel stellt eine ziemlich direkte Virtualisierung der unterliegenden Hardware via Capability geschützten Abstraktion dar. Daten und Capabilities werden in Seiten, deren Grösse bei der Hardware bestimmt ist, gespeichert. Capabilities werden zudem auch in sogenannten *nodes* gespeichert, wobei ein *node* 32 Capabilities speichern kann. Die Adressräume werden in Bäume aufgebaut, d.h. sie bestehen aus den zuvor genannten nodes. Deren Blätter sind Daten- und Capability-Pages. Durch die Nutzung eines Baumes für das Adressraummapping ist es möglich eine fein gradige Spezifikation von fault Handlern vorzunehmen. Das traversieren des Baumes ist aber ziemlich teuer, weshalb einige Optimierungen notwendig sind.

Der Kernel exportiert die Prozessabstraktion in Anwendungscode via zwei Typen von Capabilities:

- *Process Capabilities* welche die Operationen zum Manipulieren des Prozesses liefern
- *Entry Capabilities* welche dem Halter das aufrufen der Dienste eines Programms, innerhalb eines bestimmten Prozesses, erlaubt.

EROS unterscheidet nicht zwischen Prozessen und Threads. Alle Prozesse sind single-threaded. Zwei Prozesse können evtl. einen Adressraum sich teilen, jedoch können beide verschiedene Capabilities besitzen. Jeder Prozess ist eine *protection domain*. Jeder Ressourcenzugriff in EROS wird via einer *Capability invocation* vollbracht. Wenn es bei der Capability autorisiert ist, löst jeder Aufruf bei dem genannten Objekt eine Objekt definierte Operation, die vom Aufrufer spezifiziert wurde, aus. Es existiert ein einheitliches Interface, was bedeutet das alle Capabilities die selben Argumente annehmen, was ein Vorteil bei der Performanz bringt.

Die Korrektheit des EROS Betriebssystem liegt in dem Fakt das Operationen die vom Kernel ausgeführt werden in einem korrekten Systemzustand resultieren, solange der Initialzustand korrekt war. Die Herausforderung in der Kausalen-Ordnung ist zu garantieren das das wiedererlangen eines korrekten Systemzustands, nachdem das System ungeplant Heruntergefahren ist z.B. durch ein crash, möglich ist. Um das zu erreichen muss wie bereits erwähnt ein konsistentes Abbild des Systems erstellt werden. EROS Snapshot-Mechanismus führt zuvor einen Konsistenz Check durch, bevor der Snapshot durchgeführt wird, anders als bei z.B. L3. Falls der Test fehlschlägt wird das System neu gestartet ohne den derzeitigen Checkpoint zu speichern.

Die EROS IPC Implementierung wurde stark von L3 und Mach 4 beeinflusst. Zudem besitzt L3 auch ein Persistierungsmechanismus der aber nicht so ausgereift ist wie der von EROS.

Laut den Autoren [5] handelt es sich bei den Capabilities vom Exokernel um keine Capabilities sondern eher ähnlich zu dem erweiterbaren uid/gid Abstraktion, da sie keine Objekte benennen oder direkt Zugriff autorisieren.

Die Forschung an dem EROS Betriebssystem wurde bereits eingestellt. Ein Ableger des EROS Betriebssys-

tems existiert und nennt sich CapROS, welches die Codebasis benutzt [17]. An der Johns Hopkins Unversität wird zudem an einem nachfolge Projekt geforscht genannt Coyotos. Dieses versucht einige Architekturdefizite von EROS zu beseitigen [18].

E. Asbestos

Asbestos ist ein neuer Prototyp eines Betriebssystems, welches neuartige Kennzeichnungs- und Isolierungsmechanismen liefert die helfen *exploitable* Software einzugrenzen.

Ein mächtiges Werkzeug um *exploits* einzugrenzen ist das Prinzip des geringsten Rechts. D.h. das jede Systemkomponente die minimale Anzahl an Rechten besitzen soll die es benötigt um seine Aufgabe auszuführen. Siehe dazu auch den Ansatz von CAP in Abschnitt II-C. Unglücklicherweise können derzeitige Betriebssysteme nicht das Prinzip des geringsten Recht erfüllen. Asbestos Beiträge sind zweifälltig, zum einen benutzen alle Zugriffskontrollen die Asbestos labels. Es handelt sich dabei um eine Primitive die die Vorteile von der discretionary (DAC) und mandatory (MAC) Zugriffskontrolle kombiniert.

Bei DAC wird die Entscheidung, ob auf eine Ressource zugegriffen werden darf, allein auf der Basis der Identität des Akteurs getroffen [19]. Im Gegensatz zu MAC wo die Entscheidungen über Zugriffsberechtigungen nicht nur auf der Basis der Identität des Akteurs (Benutzers, Prozesses) und des Objekts (Ressource, auf die zugegriffen werden soll) gefällt, sondern aufgrund zusätzlicher Regeln und Eigenschaften (wie Kategorisierungen, Labels und Code-Wörtern) gemacht werden [20].

Die Labels bestimmen welche Dienste ein Prozess aufrufen und mit welchen anderen Prozessen er interagieren kann. Asbestos Labels können zudem den Informationsfluss in System- und Anwendungskomponenten verfolgen und limitieren. Sie unterstützen dezentralisierte Abteilungen, die jeder Prozess dynamisch erstellen und manipulieren kann. Asbestos gibt einen Programm das eine neue Abteilung erstellt ein willkürliches Recht, um die Daten in der Abteilung freizugeben. Das Programm kann dieses Recht weggeben, welches das Recht ähnlich einer Capability macht.

Der zweite Beitrag Asbestos ist die *event process* Abstraktion, welche für Serveranwendungen effizient viele gleichzeitige Nutzer unterstützt und isoliert. In normalen Label-Systemen würden Serverprozesse schnell, durch Daten die zu mehreren Benutzern gehören, Kontaminiert werden und die Möglichkeit verlieren auf jeden zu antworten. Event Prozesse erlauben es, einen einzelnen Prozesszustand privat für mehrere Benutzer zu behalten. Aber diesen Zustand so zu isolieren, sodass ein *exploit* nur die Daten des Nutzers beeinflussen würde. Eine Gruppe von Event Prozessen ist genauso effizient als ein einzelner gewöhnlicher Prozess [6, S. 1].

Asbestos IPC ähnelt dem Mikrokernel IPC wie z.B. von Mach. Prozesse kommunizieren mithilfe von Nachrichten die an Ports gesendet werden. Ein Prozess kann beliebig viele Ports erstellen. Nachrichten gesendet an Ports werden an ein einzelnen Prozess gesendet mit Empfangsrechten für diesen Port. Initial hat der Prozess die Empfangsrechte der den Port erstellt, aber diese Rechte sind übertragbar. Das Nachrichtenprotokoll von Asbestos wurde von 9P (Plan 9) inspiriert, siehe Abschnitt II-B2.

Durch die Integrierung der Kommunikationsports mit dem Label System wird nicht nur die unerwünschte Kontamination verhindert sondern die Semantik von Capability basierende Senderechten erreicht. Das Resultierende Port-Label System unterstützt also Capability ähnliche Senderechte. Genauso wie bei den Prozessen beschränkt der Kernel die Privilegien der Event Prozesse, während diese die ankommenden Nachrichten für den Nutzer behandeln und isoliert verschiedene Event Prozesszustände. Jeder Event Prozess ist einem konventionellen *base* Prozess assoziiert. Der Kernelzustand der Event Prozesse besteht aus einem *send*, *recieve* Label, *recieve rights* für bestimmte Ports und eine Menge privater Speicherseiten plus einige Buchhaltungsinformationen. Alles zusammen zählt 44 Bytes vom Asbestos Kernelspeicher. Zum Vergleich Asbestos kleinste Prozessstruktur nimmt 320 Bytes ein.

Der *base* Prozess wird nachdem aufrufen des sogenannten *ep_checkpoints* de-scheduled bis eine Nachricht auf einem Port ankommt an dem der *base* Prozess oder einer der Event Prozesse *recieve* Rechte besitzt. Die Nachricht wird dann behandelt und danach wird der Event Prozess wieder suspendiert. Diese Prozessvariante verursacht weniger Overhead als wenn der Adressraum aufgeteilt wird somit können viele tausende Event Prozesse theoretisch Co-existieren.

IV. Fazit

Viel Arbeit ist in sichere System bereits geflossen, vor allem wurde schon sehr früh über Sicherheitsmechanismen nachgedacht, jedoch sind sie oft an der Umsetzung oder Nutzbarkeit gescheitert. Zusammenfassend ist zu sagen das einzig die Ansätze von Unix bzw. Plan 9 und Mikrokernel sich wirklich in die heutigen Systeme durchgesetzt haben. Betrachten wir die heutigen Zugriffsmechanismen so werden in Linux und deren

Variationen weiterhin Access Bits und in Windows oder Mac OS X sogenannte *access control lists* (ACL) verwendet [21], [22]. In Linux können zudem ACLs benutzt werden, sodass später ein Datenaustausch mit Windows über Samba vollzogen werden kann [23].

Das die anderen Systeme sich nicht durchgesetzt haben heißt jedoch nicht das deren Ansätze schlecht waren. Das Problem was oft sichere Systeme besitzen ist wie bereits erwähnt die Performanz oder Nutzbarkeit. Was für viele normale Nutzer ein absolutes K.O. Kriterium ist, da die meisten nicht besonders viel auf Sicherheit setzen. Jedenfalls nicht soviel das sie einige Defizite annehmen. Dies ist durch aus wahrscheinlich eines der Gründe warum sich solche Forschungssysteme nicht durch setzen konnten, davon abgesehen das andere Systeme heutzutage den Markt beherrschen und die Unterstützung für sichere Systeme nicht groß genug ist.

LITERATUR

[1] I. S. on Security & Privacy, "Systematization of knowledge frequently asked questions," 2011, besucht am 09.04.2014. [Online]. Available: http://www.ieee-security.org/TC/SP2011/sokfaq.html

[2] J. Liedtke, "On micro-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95*. New York, NY, USA: ACM, 1995, pp. 237–250.

[3] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole, "Exokernel: An operat- ing system architecture for application-level resource management." in *SIGOPS Oper. Syst. Rev., 29(5)*, December 1995, pp. 251–266.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the spin operating system," in *SIGOPS Oper. Syst. Rev., 29(5)*, December 1995, pp. 267–283.

[5] J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: A fast capability system," in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP '99*. New York, NY, USA: ACM, 1999, pp. 170–185.

[6] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazi'eres, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *SIGOPS Oper. Syst. Rev., 39(5)*, October 2005, pp. 17–30.

[7] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The multics virtual memory," in *Proceedings of the Second Symposium on Operating Systems Principles, SOSP '69*. New York, NY, USA: ACM, 1969, pp. 30–42.

[8] M. D. Schroeder and J. H. Saltzer, "A hardware architecture for implementing protection rings," in *Commun. ACM, 15(3)*, March 1972, pp. 157–170.

[9] D. M. Ritchie and K. Thompson, "The unix time-sharing system," in *Proceedings of the Fourth ACM Symposium on Operating System Principles, SOSP '73*. New York, NY, USA: ACM, 1973, pp. 365–375.

[10] R. Pike, D. L. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from bell labs," in *Computing Systems*, 1995, pp. 221–254.

[11] R. M. Needham and R. D. Walker, "The cambridge cap computer and its protection system," in *SIGOPS Oper. Syst. Rev., 11(5)*, November 1977, pp. 1–10.

[12] Linode, "Linux users and groups," 2014, besucht am 09.04.2014. [Online]. Available: https://www.linode.com/docs/tools-reference/linux-users-and-groups

[13] Computerhope, "Linux and unix groupmod command," 2015, besucht am 09.04.2014. [Online]. Available: http://www.computerhope.com/unix/groupmod.htm

[14] T. Mucke and M. Petter, "Linux - kurzeinführung auf 200 seiten," 2002, besucht am 09.04.2014. [Online]. Available: http://linuxhandbuch.sourceforge.net/html/linuxhandbuch.htmlch12.html

[15] D. of Computer Science and E. U. of Washington, "Spin papers," 1997, besucht am 09.04.2014. [Online]. Available: http://www-spin.cs.washington.edu/Distro/docs/papers/index.html

[16] ——, "The spin operating system distribution site," 1997, besucht am 09.04.2014. [Online]. Available: http://www-spin.cs.washington.edu/Distro/index.html

[17] S. D. Group, "Overview of capros: Architecture and benefits," 2009, besucht am 09.04.2014. [Online]. Available: http://www.capros.org/overview.html

[18] L. The EROS Group, "The coyotos secure operating system," 2010, besucht am 09.04.2014. [Online]. Available: http://www.coyotos.org/

[19] F. B. Schneider, "Access control," 2012, besucht am 09.04.2014. [Online]. Available: https://www.cs.cornell.edu/fbs/publications/chptr.DAC.pdf

[20] ——, "Mandatory access control," 2014, besucht am 09.04.2014. [Online]. Available: https://www.cs.cornell.edu/fbs/publications/chptr.MAC.pdf

[21] MSDN, "Access control," 2015, besucht am 09.04.2014. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa374860(v=vs.85).aspx

[22] V. Danen, "Introduction to os x access control lists (acls)," 2011, besucht am 09.04.2014. [Online]. Available: http://www.techrepublic.com/blog/apple-in-the-enterprise/introduction-to-os-x-access-control-lists-acls/

[23] S. Linux, "Access control lists unter linux," 2003, besucht am 09.04.2014. [Online]. Available: http://users.suse.com/\textasciitildeagruen/acl/chapter/fs_acl-de.pdf