Stackable File Systems

Robert Jankovics, am 05.11.2004
e0125997@student.tuwien.ac.at

Kurzfassung
Traditionelle Dateisystem Entwicklung ist ein langsamer, aufwendiger Prozess. Aufgrund der engen Verbindung zwischen Betriebsystem und Dateisystem, und der Tatsache dass Dateisysteme ein abgeschlossenes System bilden hat jede Weiterentwicklung nur dann Bestand, wenn sie von den großen Betriebsystem Herstellern getragen wird.
Das Design der Stackable File Systems führt weg von einem abgeschlossenen System hin zu einer Architektur aus unterschiedlichen, unabhängigen Layern welche von beliebigen Drittanbietern kommen können.
Die Implementierung solcher Layer fordert jedoch komplexen, plattformabhängigen low-Level Code. Hier schafft FiST, eine Programmiersprache zur Beschreibung und Erstellung von File Systemen, Abhilfe. FiST ermöglicht plattformunabhängigen Code auf einem hohen Abstraktionsniveau.

1. Einleitung

Das Dateisystem ist eines der wichtigsten Betriebsystem Services dass dem Benutzer durch bereitstellen verschiedenster Funktionen den Umgang mit Daten jeglicher Art ermöglicht und vereinfacht. Dies beginnt bei der Bereitstellung einer Möglichkeit Dateien in Verzeichnissen zu strukturieren und reicht bis hin zur Rechteverwaltung solcher Verzeichnisse bzw. einzelner Dateien. Durch hinzufügen weiterer Funktionalitäten könnte die Effizienz und die Benutzerfreundlichkeit eines solchen Systems weiter gesteigert werden. Trotzdem werden File Systeme oft über Jahre hinweg nicht weiterentwickelt. Und obwohl verschiedenste Ideen für innovative Erweiterungen vorhanden und teilweise auch in manchen Labors implementiert worden sind, hält dieser Zustand fortdauernd an. Wirft man hingegen einen Blick auf den Anwendungssektor erkennt man eine unglaubliche Geschwindigkeit mit welcher neue Anwendungen entstehen und weiterentwickelt werden. Der Grund hierfür liegt in der großen Anzahl an Softwareherstellern woraus sich ein produktiver Konkurrenzkampf ergibt.
Stackable File Systems [1] basieren auf einer Layer Architektur wobei jeder Layer ein erweiterbares symmetrisches Interface zur Verfügung stellt. Syntaktisch gesehen können die verschiedenen Layer in einer beliebigen Reihenfolge angeordnet werden. Die Position eines einzelnen Layers ist lediglich durch einen möglichen semantischen Zusammenhang mit anderen Layern festgelegt. Einzelne Layer können einem Dateisystem beliebig hinzugefügt oder entfernt werden. Die gesamte Funktionalität eines so entstandenen Dateisystems setzt sich somit aus der Gesamtheit der Einzelfunktionalitäten der beteiligten Layer zusammen. Dies bietet verschiedensten Drittanbietern die Möglichkeit eigene Services zu entwickeln welche nach dem "Plug in" Prinzip zu einem bestehenden Dateisystem hinzugefügt werden können. Dadurch sollte auch hier ein Konkurrenzkampf entstehen der dazu führt, dass ähnlich wie auf dem Anwendungssektor die Entstehung neuartiger Produkte in erhöhtem Tempo vorangetrieben wird.
Vorstellbar wären Layer für Dateisystem Dienste wie in etwa:

- automatische Kompression und Dekompression
- automatische Ver- und Entschlüsselung
- rückgängig machen u. wiederherstellen
- individualisiertes und somit effektiveres Caching
- selektives Daten Backup
- Transaktionen

Die Erstellung diverser Layer erfordert jedoch eine Menge komplexen low-Level Code und, eine sehr genaue Kenntnis des zu Grunde liegenden Betriebsystems um die nötigen Anpassungen an den richtigen Stellen durchführen zu können. Hier kann FiST eingreifen und weiterhelfen. FiST ist eine Programmiersprache für die Erstellung neuer File System Layer auf einer hohen Abstraktionsebene. Der geschriebene Code wird von einem FiST Compiler in File System Module übersetzt, wobei der Compiler auf die Betriebsystem spezifischen Eigenheiten Rücksicht nimmt und somit den Entwickler um die Notwendigkeit der genauen Betriebsystem Kenntnis erleichtert. Durch FiST entsteht somit einfacher kompakter Code der plattformübergreifend eingesetzt werden kann.
In Kapitel 2 dieser Arbeit wird auf den Unterschied zwischen herkömmlichem File System Design und Stackable File System Design eingegangen. Kapitel 3 und 4 beschäftigen sich in weiterer Folge mit verschiedenen Techniken der Layer Entwicklung und deren Implementierung. In Kapitel 5 wird anhand von FiST der effektive Einsatz der Stackable File Systems demonstriert indem Beispiele aufgezeigt werden um danach anhand derer die Effizienz dieser Sprache zu bestimmen.

2. File System Design

Im Folgenden werden die Unterschiede zwischen traditionellem File System Design und Stackable File System Design aufgezeigt um zu erkennen worin die größten Vor- bzw. Nachteile liegen.

2.1 Traditionelles File System Design

Die nahe liegenste Möglichkeit einem Dateisystem eine neue Funktionalität hinzuzufügen ist ein bestehendes System zu ändern. Der große Nachteil dabei liegt auf Der Hand, für jedes Betriebsystem muss jede Änderung neu implementiert werden. Weiters sind Änderungen an einem bestehenden Dateisystem sehr schwierig da die richtigen und vor allem alle Stellen im vorhandenen Code zu finden sind die angepasst werden müssen. Um die Korrektheit eines geänderten Systems zu verifizieren muss danach ein großer Teil des Codes durchgesehen werden. Die Voraussetzung hierfür ist allerdings die Verfügbarkeit des jeweiligen Source Codes. Bei Quellcode Längen von bis zu 30.000 Zeilen C Code [6] lässt sich der nötige Aufwand erahnen.
Anstelle von Änderungen auf Kernel Ebene können auch Veränderungen auf Benutzerebene durchgeführt werden was eine wesentliche Erleichterung in Hinblick auf Entwicklung und Portabilität mit sich bringt [2]. Der Nachteil liegt hier aber in der Performance Einbußung den der zusätzliche Prozessoverhead mit sich bringt. Der Performanceverlust kann in einem Bereich von bis zu einer Zehnerstelle liegen [3, 4].

Teilweise Abhilfe können vom Dateisystem gebotene standard Schnittstellen, wie zum Beispiel das Vnode Interface von UnixFS [5], schaffen.
Das Vnode Interface wird auch zur Implementierung von Stackable File Systems genutzt. Der Unterschied zum traditionellen Design liegt in der Bereitstellung eines erweiterbaren Interface anstelle von benutzerspezifischen Ergänzungen ohne Erweiterungsmöglichkeit für Außenstehende.

2.2 Stackable File System Design

Wie bereits erwähnt besteht ein Stackable FS aus verschiedenen Layern, wobei jeder Layer ein symmetrisches Interface zur Verfügung stellt was zur Folge hat, dass die syntaktische Anordnung der einzelnen Schichten beliebig ist. Einzig semantische Bedingungen schränken die Reihenfolge der Layer ein. (Es macht zum Beispiel keinen Sinn wenn ein Layer der die

Funktionalität eines Backups bietet vor einem Layer für Verschlüsselung steht, da sonst alle Daten unverschlüsselt gesichert werden.)
Funktionen werden also durch Layer die diese implementieren hinzugefügt, wobei jeder Layer nur die Operationen ausführt für die er vorgesehen ist. Man kann sich das ganze als eine Art Verarbeitungskette vorstellen mit einem Informationsfluss von oben nach unten und, einem von unten nach oben. Ein Funktionsaufruf wird von einem zum nächsten Layer weitergegeben bis der unterste Layer erreicht ist. Von dort aus werden eventuell angeforderte Daten oder Meldungen wieder nach oben hin von Layer zu Layer weitergereicht. Ein Layer kann nun in diesen Informationsfluss eingreifen und ihn verändern, oder wenn er nicht für die angeforderte Funktion zuständig ist den Fluss unverändert an seinen Nachfolger weiterleiten. (z.B. wird ein Layer der für die Verschlüsselung von Daten zuständig ist nur bei Schreibzugriffen eingreifen, wogegen für den Layer der die Entschlüsselung realisiert nur Lesezugriffe von Interesse sind).

3. Stackable Layering Techniken

Dieser Abschnitt beschäftigt sich mit verschiedenen Techniken zur Strukturierung von Layer.

3.1 Layer Komposition

Die Idee der Stackable File Systems ist die Entwicklung von eigenständigen, unabhängigen Layern. Dies ermöglicht nicht nur das Hinzufügen und Entfernen beliebiger Layer sondern erhöht die Wiederverwendbarkeit und vereinfacht das Debuggen und Fehlereingrenzen. Dies fordert eine gute Planung der Aufteilung der Funktionen eines FS. Möchte man ein Unix File System mit Datenkomprimierung implementieren, könnte eine möglich Layeraufteilung wie folgt aussehen: directory Layer – compression Layer – Unix file Layer – disk device

3.2 Layer Substitution

Unter Layer Substitution (Layer Substitution) versteht man das austauschen semantisch gleichbedeutender Layer. Semantisch gleichbedeutende Layer wären zum Beispiel ein Unix file Layer und ein log-FS Layer.

3.3 Nonlinear Stacking

In der Regel sind File System Stacks linear, was bedeutet dass jeder Zugriff von oben nach unten abgearbeitet wird. Gelegentlich bevorzugt man aber nichtlineare Stacks. Mit Fan-out wird in diesem Zusammenhang die Anzahl der Nachfolge-Layer eines Layers beschrieben. (siehe Abbildung 1). In einem verteilten Dateisystem wird ein Layer benötigt der den Zugriff auf ein FS eines anderen Computers erlaubt. Werden aber Daten angefordert die lokal liegen wird der remote Zugriff nicht benötigt. Der logical FS Layer hat einen Fan-out von 2.

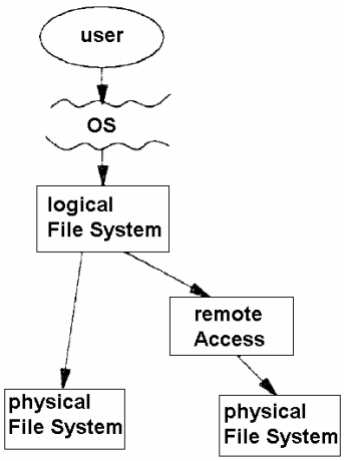
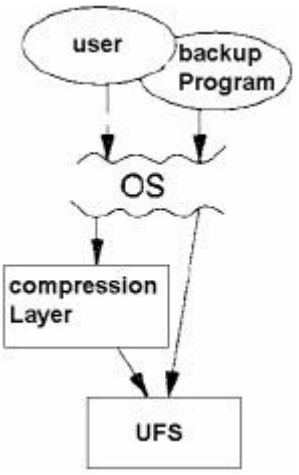


Abb. 1: Fan-out

Mit Fan-in wird die Anzahl der Vorgänger eines Layers beschrieben.
Über einen eigenen Einstiegspunkt könnte zum Beispiel der Netzwerkzugriff auf verschlüsselte Daten durch einen direkten Zugriff auf den zugrunde liegenden Verschlüsselungslayer erfolgen, womit die Notwendigkeit der Klartext Übertragung entfallen würde.



In Abbildung 2 ist ein weitere Anwendungsfall ersichtlich bei dem ein Backupprogramm direkt auf die komprimierten Daten zugreifen kann.

Abb. 2: Fan-in

3.4 Kooperierende Layer
Kooperierende Layer (cooperating Layer) sind Layer die eine Aufgabe zusammen erfüllen. Die in Abbildung 2 sichtbaren physical FS Layer sind kooperierende Layer.

3.5 Kompatibilität zwischen Layern
Die Weiterentwicklung von einzelnen Layern führt oft zu einer notwendigen Änderung des Interface. Dies führt ersichtlicher Weise zu Kompatibilitätsproblemen zu älteren Versionen. Um solche Probleme in den Griff zu bekommen können dünne Kompatibilitäts Layer eingefügt werden die die Unterschiede zwischen verschiedenen Versionen managen. Die Idee der Kompatibilitäts Layer wird manchmal auch eingesetzt um verschiedene spezifische Betriebsystem Eigenschaften zu handhaben.

4. Implementierung
Dieser Abschnitt beschäftigt sich mit den Fragen wie Layer miteinander verknüpft werden und wie die Erweiterung gewährleistet wird.

4.1 Standard File System Interfaces
Ein gutes Beispiel eines solchen Interface ist Sun's Vnode Interface [5] das für das BS Unix entwickelt wurde. Das Vnode Interface bietet die Abstraktion der File System Implementierung in Hinsicht auf Kernel Eigenheiten. Im Vnode Interface wird Datenzugriff über zwei abstrakte Datentypen realisiert. Es gibt die sog. Vnodes die einzelne Dateien oder Verzeichnisse identifizieren. Eine Datei liefert ein nicht interpretiertes Byte-Array an Daten. Ein Verzeichnis liefert eine Liste an Dateien und referenziert weitere Verzeichnisse wodurch eine Baumstruktur aufgebaut wird. Der andere abstrakte Datentyp heißt vfs. Er repräsentiert Gruppen von Dateien und Verzeichnissen, diese werden normalerweise File System oder Partition genannt. Durch mounten wird ein neuer Subtree zu einem globalen File System

Namespace hinzugefügt. Jeder Subtree kann seine eigene Implementierung haben (sein eigenes File System).
Durch den Mount Mechanismus können Stackable File Systems realisiert werden. Layer werden in Form von Subtrees zusammen gemountet. Jeder mount Befehl fügt einen neuen Layer hinzu. Normaler Weise wird der Baum von unten nach oben aufgebaut, so dass bei jedem mount Befehl der Vorgängerlayer angegeben werden kann.

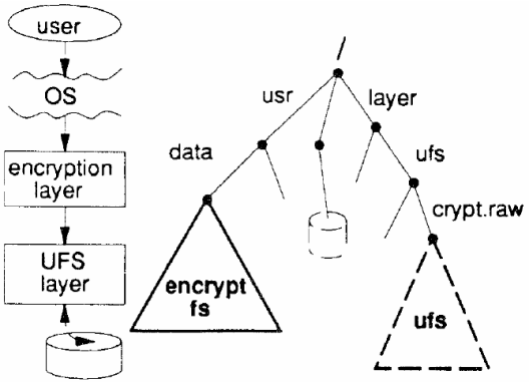


Abb. 3: Ein Verschlüsselungslayer wird auf ein UFS gemountet

In Abbildung drei wird an die Stelle crypt.raw ein UFS (Unix FS) gemountet.
Danach wird ein Verschlüsselungslayer gemountet der als Vorgänger-Layer das UFS angibt.

Während ein Stack auf dem Level eines Subtrees aufgebaut wird werden die meisten Anwender Aufrufe auf einzelnen Dateien ausgeführt.
Eine Datei wird durch Vnodes repräsentiert, pro Layer existiert ein Vnode.

4.2 Stack Daten Cachen
Um die Performance zu erhöhen möchten verschiedene Layer ihre Daten cachen. Würde jeder Layer seine Daten einzeln cachen könnte es passieren, dass verschiedene Layer auf verschiedenen Cache-Kopien arbeiten. Um das zu verhindern wird ein Cache Manager benötigt der das Cachen koordiniert.
Die Idee eines zentralen Cache Managers wurde 1994 erstmals von Heidemann und Popek [6] aufgeworfen. Bevor ein Layer Daten cachen kann muss er diese bei dem Cache Manager anfordern. Wurden die entsprechenden Daten zuvor noch niemals angefordert kann der auffordernde Layer sie sofort cachen. Ansonsten wird der Layer welche die Daten zuvor angefordert hat dazu aufgerufen diese zu aktualisieren.
Ein Jahr später verfassten Heidemann und Popek eine eigene Arbeit über dieses Thema [7].

4.3 Erweiterbarkeit
Einer der größten Vorteile den ein Stackable File System bringt ist die Möglichkeit neue Funktionalität durch hinzufügen neuer Layer zu erreichen. Oft verändern Layer in der Mitte des Stacks nur wenige Operationen und parsen den Rest einfach an den Nachfolger weiter. (zB. gibt ein Ver-Entschlüsselungs Layer Verzeichnismanipulationsoperationen an seinen Nachfolger weiter ohne sonst etwas zu machen). Es ist ersichtlich dass dieser Layer in der Mitte nicht alle Operationen der anderen Layer kennen kann. Dies wäre gar nicht möglich, da sonst jeder Layer verändert werden müsste wenn ein neuer Layer mit einer neuen Operation dem Stack hinzugefügt wird. Daher muss es eine Möglichkeit geben unbekannte Operationen mit ihren Attributen durchzuschleusen. Dies wird oft als Bypass Routine bezeichnet.

5. Einsatz anhand von FiST
Die Entwicklung von stackable File Systems ist in der Regel eine aufwendige und System spezifische Angelegenheit, da eine große Menge an komplexen Kernelcode geschrieben werden muss. Dateisysteme die im Kontext des Benutzers laufen sind wesentlich leichter zu Entwickeln, aufgrund des häufigen Kontext Wechsels ist der Performance Verlust allerdings zu groß. Die Portierung auf ein anderes System wäre aber wesentlich einfacher.

Um beiden Anforderungen gerecht zu werden wurde FiST (a File System Translator language), eine Sprache zur Erstellung von System unabhängigen Dateisystemen, erstellt. Es existieren drei Vorteile die der Einsatz einer solchen Sprache mit sich bringt:

1. **Einfachheit**: Eine Sprache für Dateisysteme bietet einen höheren Grad der Abstrahierung und vereinfacht damit die Entwicklung da Programmierer nicht mehr über komplexen low level Code bescheid wissen müssen. Sie kann sinnvolle standard Vorgaben definieren und reduziert damit die Mengen an Code die der Entwickler erstellen muss.
2. **Portierbarkeit**: Die Beschreibung der Dateisystem Funktionalität muss lediglich einmal erstellt werden. Der Compiler kümmert sich in weiterer Folge um die Betriebsystem Eigenschaften. Damit kann für jedes System für welches ein Compiler existiert ein Dateisystem aus der einmal erstellten Beschreibung generiert werden.
3. **Spezialisierung**: Die Sprache erlaubt es dem Entwickler das Dateisystem seinen Anforderungen nach zu gestalten. Anstatt eines großen komplexen Systems das nach eigenen Bedürfnissen konfiguriert werden muss kann ein schlankeres System mit genau den notwendigen Funktionen erstellt werden. Das erhöht die Performance und reduziert den Speicherbedarf. Eigens entwickelte Dateisysteme enthalten nur notwendigen Code.

Der FiST Compiler (fistgen) verwendet bei der Erstellung von Dateisystemen ein Template (Basefs). Dieses Basefs Template implementiert ein minimales stackable File System. Es fügt dem zugrunde liegenden Dateisystem keinerlei Funktionalität hinzu, es werden lediglich stacking Funktionen bereitgestellt. Die Hauptaufgabe dieses Templates ist die Handhabung vieler Kernel spezifischen Details für stacking Funktionen und die Bereitstellung sogenannter hooks in denen später von fistgen der vom Entwickler geschriebene Code eingefügt wird.

5.1 Design

FiST ist eine Sprache die das Dateisystem auf einem hohen Niveau abstrahiert. Sie abstrahiert Betriebsystem übergreifend den Blick auf verschiedenen Variationen des Vnode Interfaces.

Folgende Abbildung zeigt das Zusammenspiel der drei Komponenten des FiST Systems:

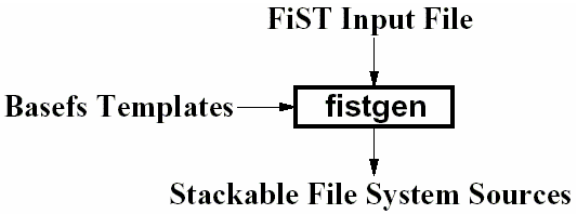


Abb. 4: FiST Komponenten

Der Entwickler erzeugt ein FiST input file in dem die Funktionalität des zu erstellenden Dateisystems beschrieben ist. Fistgen, der FiST Sprachen Code Parser und Dateisystem Generator liest diese Eingabe Datei. Zusätzlich zu der Eingabedatei benötigt der Compiler das Basefs Template. Dieses stellt die stacking Funktionalität jedes Betriebsystems bereit und enthält bestimmte Stellen (hooks) die als Platzhalter fungieren. Fistgen kombiniert das Template und die Eingabedatei und erzeugt Kernel C Code der das neue Dateisystem darstellt. Der Entwickler kann des Weiteren etliche Dateisystem Features ein- bzw. ausschalten (z.B. Fan in). Fistgen erzeugt abhängig davon den benötigten C Code.

5.2 Ein Beispiel - Snoopfs

Im Folgenden wird der Einsatz von FiST anhand eines einfachen Beispiels demonstriert. Um die Vorteile dieser Sprache besser zum Ausdruck bringen zu können wird die Erstellung mit der traditionellen Dateisystem Entwicklung verglichen.

Mit diesem Beispiel soll ein Dateisystem erstellt werden, das herumspionierende Leute (snooping people) aufdecken soll. Die Idee dahinter ist, dass nur der Besitzer einer Datei oder der Root Benutzer diese lesen dürfen. Normaler Weise würde eine nicht berechtigte Person eine „Zugriff verweigert“ Meldung bekommen. Es wird aber in keiner Weise festgehalten wer wann und wie oft versucht ohne Berechtigung auf Dateien zuzugreifen. Snoopfs soll diese Versuche mitprotokollieren.

Die Stelle an der diese Überprüfung stattfinden soll ist die lookup Routine die genutzt wird um eine Datei in einem Verzeichnis zu finden. Dies würde ohne den Einsatz von FiST folgende Schritte verlangen:

1. Es muss ein Betriebsystem gefunden werden für welches der Quellcode des eingesetzten Dateisystems verfügbar ist.
2. Der Entwickler muss den Kernel Code lesen, verstehen und alle relevanten Bereiche finden.
3. Der Code muss so modifiziert werden, dass er die gewünschte Funktionalität implementiert.
4. Der geänderte Code muss neu kompiliert und das System neu gestartet werden.
5. Das neue Dateisystem muss getestet werden. Bei Bedarf müssen Fehler gesucht und behoben werden.

Wenn diese Punkte alle erledigt sind besitzt der Entwickler den modifizierten Code für ein Betriebsystem mit einem Dateisystem. Die Anzahl der zu verstehenden Codezeilen liegt dabei in den Tausenden. Dieser Vorgang muss für jedes Betriebsystem und für jedes Dateisystem wiederholt werden.

Im Vergleich dazu die Entwicklung mit FiST:

1. Der Code für das neue Dateisystem muss einmal erstellt werden.
2. Dieser Code muss mit fistgen zu ladbaren Kernelmodulen kompiliert werden.
3. Das Kernelmodul muss geladen werden.
4. Das neue Dateisystem kann gemountet und getestet werden.

Der Entwickler muss nicht mehr über Kernel spezifische Details bescheid wissen. Das Dateisystem muss nur einmal geschrieben werden und ist auf andere Systeme portierbar. Der vorhandene Kernel muss nicht neu kompiliert und das System muss nicht neu gestartet werden. Der zu produzierende Code hat eine Länge von sieben Zeilen:

```
%op:lookup:postcall {
if ((fistLastErr() == EPERM || fistLastErr() == ENOENT) &&
     $0.owner != %uid && %uid != 0)
   fistPrintf(“snoopfs detected access by uid %d,\pid %d, to file %s\n“,
              %uid, %pid, $name);
}
```

Dieses kurze Programm fügt nach dem normalen Aufruf der lookup Routine ein If statement ein. Es überprüft ob der zuvor gehende lookup Aufruf mit einem Fehlercode endete, wer der Besitzer des Verzeichnisses ist und wer der aktuelle Benutzer ist. Danach wird bei Bedarf eine Warnmeldung geschrieben.

Diese Beschreibung kann auf jedem System eingesetzt werden für das es einen Compiler und ein Basefs Template gibt. Zur Zeit sind das die folgenden drei Systeme: Solaris, freeBSD und Linux.

5.3 Der Aufbau der Sprache FiST

FiST ist eine high level Sprache die Dateisystem Funktionen für verschiedene Betriebsysteme anbietet. Des Weiteren ermöglicht FiST den Einsatz von C um die gewohnte Flexibilität von Systemprogrammierung zu gewährleisten. Die Eingabedatei – das FiST input file ist in 4 Bereiche gegliedert:

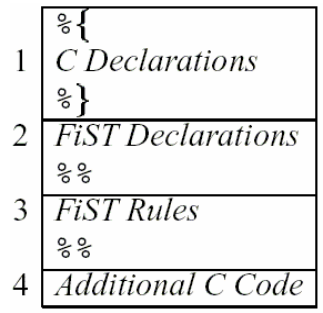


Abb. 5: FiST input file Gliederung

Die FiST Grammatik ist jener von yacc nachgebildet da yacc vielen Entwicklern bekannt ist. Die Aufteilung in vier Sektionen spiegelt das Verlangen nach vier verschiedenen Arten von Code wieder: C header Definitionen, Deklarationen die für den erzeugten Code globale Gültigkeit besitzen, Aktionen die vor/nach/während einer Vnode Operation ausgeführt werden und zusätzlicher C Code.

- C Deklarationen
  werden in "{% %}" eingeschlossen. Sie ermöglichen das Einbinden zusätzlicher C Headers, das Definieren von Makros und Typedefs, das Auflisten von forward function prototyps.
- FiST Deklarationen
  ermöglichen das Definieren globaler Dateisystem Attribute (zB Fan-in/Fan-out ist erlaubt, FS ist read only), das Definieren spezieller Datenstrukturen welche im restlichen Code verwendet werden können, das Spezifizieren von Mount Parametern (so kann z.B. einem versioning System die Anzahl der maximalen Dateiversionen in Form eines Mount Parameters übergeben werden), das Erstellen neuer Error Codes (ein Verschlüsselungs System könnte den Fehlerstatus "key expired" zurückliefern).
- FiST Regeln
  legen Aktionen fest die für bestimmte Vnode Operationen ausgeführt werden.
  Hier steht der Code der das Verhalten des Dateisystems festlegt. Es kann bestimmt werden für welche Art von Daten welche Regel ausgeführt wird, und ob diese Regel vor, nach oder anstelle des Operationsaufrufs ausgeführt werden soll.
  Die FiST Regeln bilden den primären Bereich in dem die meisten Aktionen für den zu erzeugenden Code festgelegt werden.
- Zusätzlicher C Code
  ermöglicht das Erstellen von C Funktionen welche von dem gesamten Dateisystem aus referenziert werden können. Dieser Teil wurde aufgrund von Code Modularität ausgegliedert. FiST Regeln werden auf Vnode Operationen angewandt, während C Code beliebigen Code enthält und von überall aus aufgerufen werden kann. Dieser

Bereich ermöglicht somit auch das Verwenden von bereits existierendem Code. Des Weiteren können mit C Code auch Betriebsystem spezifische Eigenheiten genutzt werden, dies resultiert allerdings in nicht portierbaren Code.

Die FiST Syntax erlaubt es gemountete Dateisysteme und deren Dateien und Attribute anzusprechen, sowie das Aufrufen beliebiger FiST Funktionen. Mount Referenzen beginnen mit `$vfs`, Datei Referenzen starten hingegen mit einem kürzeren "`$`" Zeichen da diese öfters verwendet werden. Einer Referenz folgt immer eine Zahl die die Unterscheidung zwischen verschiedenen Instanzen gewährleistet (z.B. `$1`, `$2` – wird besonders benötigt wenn Fan out genutzt wird). Attribute werden durch das Anfügen eines Punktes und dem Attributsnamen an eine Referenz angegeben (z.B. `$vfs.blocksize`, `$1.name`, `$2.owner`). Der Gültigkeitsbereich dieser Referenzen ist die aktuelle Vnode Funktion in welcher sie ausgeführt werden.
Es existiert eine Reihe an Betriebsystem Attributen auf welche nur lesender Zugriff möglich ist. Der Gültigkeitsbereich dieser Attribute auf die nur lesender Zugriff möglich ist (%) ist global:

Global	Meaning
%blocksize	native disk block size
%gid	effective group ID
%pagesize	native page size
%pid	process ID
%time	current time (seconds since epoch)
%uid	effective user ID

Tab. 1: Globale read only Attribute

FiST ermöglicht es neue Attribute zu erstellen. Möchte man zum Beispiel in einem ACL Dateisystem den Zugriff auf Dateien für bestimmte Benutzer zeitlich beschränken kann man hierfür ein neues Attribut definieren:

```
per_vnode {
    int    user;
    int    group;
    time_t expire;  /* access expiration time */
};
```

Mit oben stehender Definition können auf Benutzer und Gruppe bzw. die erlaubte Zeit mittels `$0.user`, `$0.group` und `$0.expire` zugegriffen werden. Dateien werden Attribute mittels `per_vnode`, gemounteten Dateisystemen mittels `per_vfs` zugeordnet.

FiST Code kann FiST Funktionen aufrufen. Einige dieser Funktionen werden in nachfolgender Tabelle angeführt. Der Gültigkeitsbereich dieser Funktionen ist global im gemounteten Dateisystem:

Function	Meaning
fistPrintf	print messages
fistStrEq	string comparison
fistMemCpy	buffer copying similar
fistLastErr	get the last error code
fistSetErr	set the return error code
fistReturnErr	return an error code immediately
fistSetIoctlData	set ioctl value to pass to a user process
fistGetIoctlData	get ioctl value from a user process
fistSetFileData	write arbitrary data to a file
fistGetFileData	read arbitrary data from a file
fistLookup	find a file in a directory
fistReaddir	read a directory
fistSkipName	hide a name of a file in a directory
fistOp	execute an arbitrary vnode operation

Tab. 2: Einige FiST Funktionen

Im Folgenden wird der Informationsfluss in einem stackable file system dargestellt. Ein FiST Dateisystems hat die Kontrolle über Vnode Operationen. Eine typische Vnode Operation läuft in zwei Schritten ab.

1. Finde den Vnode des Vorgänger Layers
2. Wende die selbe Operation auf diesem Layer an

Dieser Vorgang wird rekursiv bis zur letzten Ebene durchgeführt. In Layer für Layer können dabei zusätzlich zur aufgerufenen Funktion precall oder postcall Operationen ausgeführt werden.

FiST Dateisysteme können Daten die von einem Layer zum Nächsten gereicht werden in beliebiger Weise manipulieren. Dazu stellt FiST zwei Arten von Filter zu Verfügung. Wenn ein Programmierer `filter:data` im FiST input file deklariert verlangt fistgen in der C Code Sektion zwei Funktionen, `encode_data` und `decode_data`. Diese Funktionen verlangen Eingabedaten und einen allozierten Bereich gleicher Größe für die Ausgabedaten. Die beiden Funktionen müssen vom Programmiere implementiert werden. Mit diesem Mechanismus können Inhalte bzw. Namen in beliebig komplexer Weise verändert werden.

5.4 Codeerzeugung

Der FiST Code Generator fistgen liest das input File ein, wählt das richtige Basefs Template und erzeugt alle Dateien die für das neue Dateisystem notwendig sind (C source files, header files, Makefile).
Fistgen implementiert einen Teil des C Parsers und einen Teil des C Präprozessors. Er handhabt bedingte Makros (#ifdef, ,#endif) und FiST Variablen. Weiters achtet fistgen auf die Gliederung des input files und parst FiST tags in dem Basefs Template (die sog. hooks).
Nach dem Parsvorgang erzeugt fistgen interne Datenstrukturen und Symbol Tabellen für alle zu verwaltenden Schlüsselwörter. Aus diesen Strukturen, Tabellen und dem Template erzeugt fistgen das neue Dateisystem. Der erzeugte Code enthält Kommentare und ist gut leserlich. Dies ist einer der vielen Vorteile von FiST da auch noch nachträglich Änderungen durchgeführt werden können.
Das erzeugte Endprodukt sind loadable Kernel Module die das neue Dateisystem darstellen. Die Unterstützung durch das Betriebsytem dieser loadable Module ist allerdings nicht zwingend notwendig sondern nur eine Annehmlichkeit für die Entwicklung da lästige

Systemneustarts entfallen. Aber obwohl die meisten Betriebsysteme loadable Kernel Module (LKMs) zwar unterstützen bergen sie dennoch ein gewisses Sicherheitsrisiko[8].

5.5 Beispiele

In diesem Teilkapitel werden einige Beispiele für mit FiST entwickelte Dateisysteme vorgestellt. Bestimmte Schlüsselstellen dieser werden näher erläutert um einerseits besondere Fähigkeiten und andererseits die Einfachheit und Flexibilität dieser Sprache darzustellen:

5.5.1 Cryptfs

Cryptfs ist ein Verschlüsselungs-Dateisystem in dem sowohl Dateien als auch deren Name mit einem 128bit Schlüssel und dem Blowfish Algorithmus verschlüsselt werden. Nachdem Dateinamen verschlüsselt wurden werden diese noch codiert um ungültige Zeichen zu vermeiden:

```
%{
#include <blowfish.h>
%}
filter:data;
filter:name;
ioctl:fromuser SETKEY {char ukey[16];};
per_vfs {char key[16];};
%%
%op:ioctl:SETKEY {
    char temp_buf[16];
    if (fistGetIoctlData(SETKEY, ukey, temp_buf)<0)
        fistSetErr(EFAULT);
    else
        BF_set_key(&$vfs.key, 16, temp_buf);
}
%%
unsigned char global_iv[8] = {
    0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10 };
int cryptfs_encode_data(const page_t *in, page_t *out)
{
    int n = 0;        /*blowfish variables */
    unsigned char iv[8];

    fistMemCpy(iv, global_iv, 8);
    BF_cfb64_encrypt(in, out, %pagesize, &($vfs.key),
                     iv, &n, BF_ENCRYPT);
    return %pagesize;
}
…
```

Die Funktionsweise des oben stehenden Programmausschnitts wird an dieser Stelle nicht näher erläutert. Worauf hier hingewiesen werden soll sind folgende Punkte.

- In oben stehendem Beispiel sind die 4 Sektionen eines FiST Programms gut ersichtlich.
- Der Filter Mechanismus zur Verschlüsselung von Daten wird eingesetzt.

- Der verwendete Schlüssel wird mittels `per_vfs` als Attribut des Dateisystems gespeichert.
- Die Verwendung von `Ioctl`. IOcontrols werden zur Kommunikation zwischen Kernel- und Benutzer Ebene verwendet. In diesem konkreten Fall wird dem Dateisystem auf Kernel Ebene ein Schlüssel von der Ebene des Benutzers übergeben.

5.5.2 Aclfs

Aclfs erlaubt einer zusätzlichen UID und einer GID den Zugriff auf ein Verzeichnis so als wären sie der Besitzer dieses Verzeichnisses:

```
fanin no;
ioctl:fromuser SETACL {int u; int g;};
fileformat ACLDATA {int us; int gr;};
%%
%op:ioctl:SETACL {
    if ($0.owner == %uid) {
       int u2, g2;
       if  (fistGetIoctlData(SETACL, u, &u2) < 0 ||
            fistGetIoctlData(SETACL, g, &g2) < 0)
           fistSetErr(EFAULT);
       else {
           fistSetFileData(“.acl“, ACLDATA, us, u2);
           fistSetFileData(“.acl“, ACLDATA, gr, g2);
       }
    } else
       fistSetErr(EPERM);
}
%op:lookup:postcall
    int u2, g2;
    if (fistLastErr() == EPERM
        &&
        fistGetFileData(“.acl“, ACLDATA, us, u2)>=0
        &&
        fistGetFileData(“.acl“, ACLDATA, gr, g2)>=0
        &&
        (%uid == u2 || %gid ==g2))
          fistLookup($dir:1, $name, $1,
                     $dir:1.owner, $dir:1.group);
}
%op:lookup:precall {
    if (fistStrEq/$name, “.acl“) &&
        $dir.owner != %uid)
       fistReturnErr(ENOENT);
}
%op:readdir:call {
    if (fistStrEq($name, “.acl“))
       fistSkipName($name);
}
```

Hier soll auf folgende neue Merkmale hingewiesen werden:

- Fan in ist nicht erlaubt. Das macht das Dateisystem sicherer[9].
- Das bedingte Handeln aufgrund von Informationen aus speziellen .acl Dateien, bzw. das Reagieren auf eine fehlende .acl Datei.
- Das Ändern der Sicht auf das zugrunde liegende Dateisystem. Die .acl Datei ist nur für den Besitzer, nicht aber für zusätzliche Benutzer (die in der .acl Datei definiert sind) sichtbar.

Dieses Programm nutzt nur die Bereiche "FiST Deklarationen" und "FiST Regeln".

5.5.3 Unionfs

Unionfs vereint den Inhalt zweier Dateisysteme. Die zwei zugrunde liegenden Dateisysteme erscheinen dem Benutzer als ein Ganzes:

```
fanout 2;
%%
%op:lookup:postcall {
    if (fistLastErr() == ENOENT)
      fistSetErr(fistLookup($dir:2, $name));
}
%op:readdir:postcall {
    fistSetErr(fistReaddir($dir:2, NODUPS));
}
%delops:all:postcall {
    fistSetErr(fistOp($2));
}
%writeops:all:call {
    fistSetErr(fistOp($1));
}
```

5.6 Die Effizienz von FiST

Die Effektivität wird anhand dreier Faktoren bestimmt: Codelänge, Entwicklungszeit und Performance. Zur Bestimmung dieser Komponenten werden Testergebnisse der vier vorgestellten Beispiele herangezogen. Getestet wurde auf drei verschiedenen Betriebsystemen: Linux 2.3, Solaris 2.6 und FreeBSD 3.3, auf folgendem System: P5/90, 64MB RAM mit einer Quantum Fireball 4,35GB DIE Festplatte.

5.6.1 Codelänge

Die Codelänge ist ein messbarer Faktor der die notwendige Entwicklungszeit eines neuen Dateisystems beeinflusst. Um die Ersparnis unter der Verwendung von FiST zu zeigen wird der Vergleich der notwendigen Codezeilen für die vier bekannten Dateisysteme abhängig vom Entwicklungsansatz aufgezeigt. Verglichen werden folgende drei Ansätze:

1. Die Dateisysteme werden in C als Kernel Modul geschrieben
2. Die Dateisysteme werden in C geschrieben, wobei der Code des Basefs Templates als Ausgangspunkt verwendet wird.
3. Unter Verwendung von FiST.

Der Vergleich bringt folgendes Ergebnis:

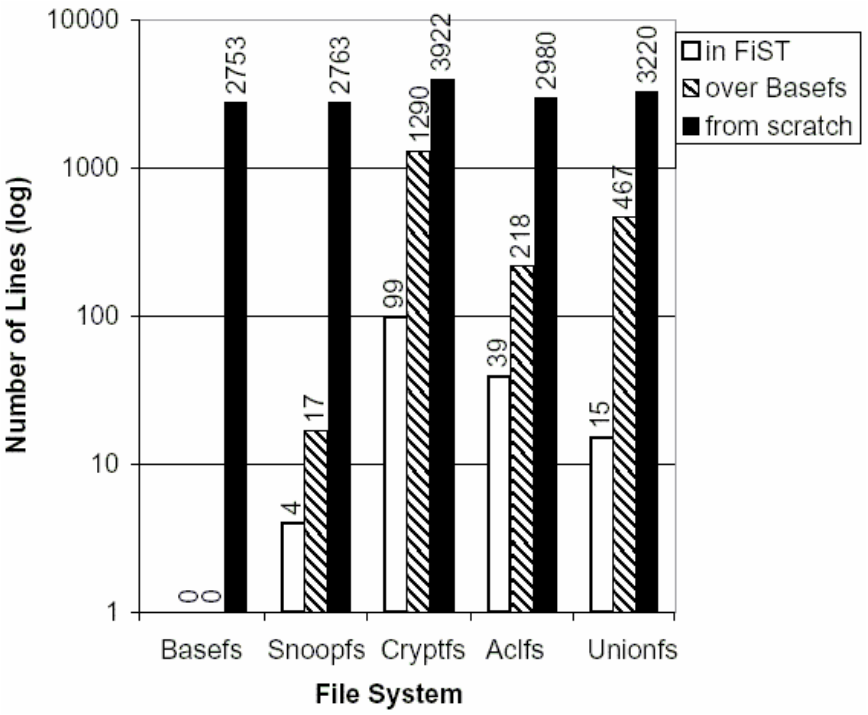


Abb. 5: Vergleich der Codelänge

Leerzeilen und Kommentare wurden bei der Zählung nicht berücksichtigt. Des Weiteren wurden bei Cryptfs 627 Zeilen des Blowfish Algorithmus nicht mitgezählt. Das Ergebnis zeigt eine drastische Reduktion der Codelänge von bis zu zwei Zehnerstellen unter der Verwendung von FiST.

5.6.2 Entwicklungszeit

Das Abschätzen der Entwicklungszeit von Kernel Software ist ein sehr schweres Unterfangen, nicht zuletzt weil das Können und die Erfahrung des Entwicklers diese Zeit wesentlich beeinflusst. Folgende quantitativen Angaben stammen von den Entwicklern der FiST Sprache.
Abbildung 6 zeigt die Anzahl der Tage die die Entwickler benötigten um die Dateisysteme für die vorher genannten drei Betriebsysteme zu erstellen:

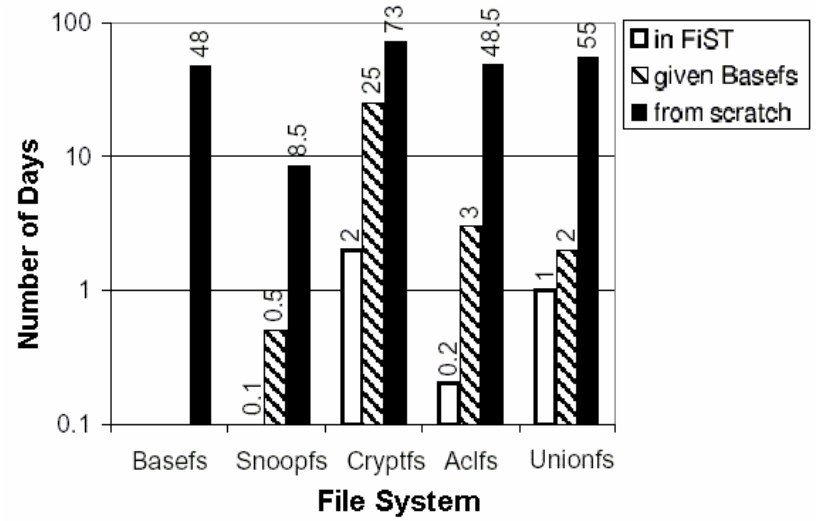


Abb. 6:Entwicklungszeiten

Die gemessenen Zeiten basieren auf der Annahme eines acht Stunden Arbeitstages.
Das Ergebnis zeigt eine Einsparung der Entwicklungszeit um einen Faktor 5-15. Dies ist auf die Tatsache zurückzuführen, dass mit FiST entwickelte Systeme nur einmal erstellt werden müssen, mit C programmierte hingegen müssen auf jedes Betriebsystem portiert werden.

5.6.3 Performance

Die Performance der mit FiST erstellten Dateisysteme wurde wieder für die drei Betriebsysteme Linux 2.3, Solaris 2.6, FreeBSD 3.3 und für die drei zugrunde liegenden Dateisysteme EXT2, UFS, FFS untersucht. Die Messung erfolgte durch das Kompilieren eines großen Packets welches 50.000 Zeilen Code in tausenden kleinen Dateien enthielt. Dieser Prozess forderte eine große Anzahl an Lese- und Schreibzugriffe sowie eine ausgeglichene Mischung aus anderen Dateisystem Operationen. Jede Messung wurde zehn Mal Aufgenommen und die Ergebnisse danach gemittelt. Der erste Durchlauf floss nicht in die Messung mit ein um den Einfluss von caching Effekten gering zu halten. Eben aus demselben Grund wurde das Dateisystem nach jeder gültigen Messung neu gemountet. Abbildung 7 zeigt den Performance Verlust in Bezug auf das zugrunde liegende Dateisystem:

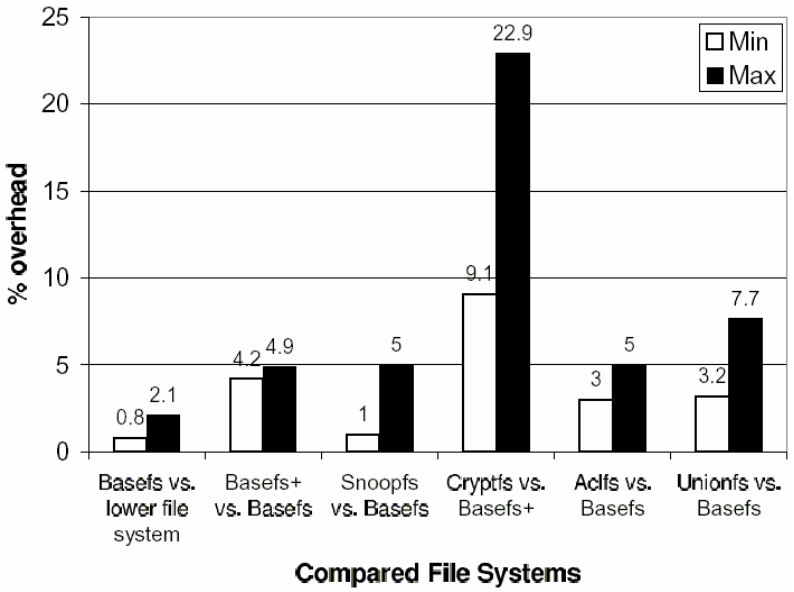


Abb. 7: Performance Verluste

Aus oben stehender Grafik lassen sich zwei Aussagen ableiten. Zum einen ist erkennbar, dass der stacking overhead alleine mit 0,8% – 2,1% klein ist. (Im Vergleich dazu sei Wrapfs genannt. Wrapfs wurde 1999 als eine Art Vorgänger von FiST durch Erez Zadok, Ion Badulescu und Alex Shender veröffentlicht[4]. Wrapfs hatte einen stacking overhead, je nach Betriebsystem von 3%-10%.)
Des Weiteren ist der Vorteil von bedingter Codeeinbindung am Vergleich zwischen Basefs und Basefs+ gut zu erkennen. Basefs+ enthält gegenüber Basefs zusätzlichen Code zur Manipulation von Dateinamen und Daten. Während Cryptfs diesen zusätzlichen Code benötigt ist das bei Snoopfs, Aclfs und Unionfs nicht der Fall. Wie früher beschrieben fügt fistgen zusätzlichen Code (z.B. zur Datenmanipulation) nur dann ein wenn er tatsächlich gebraucht wird.

Der heraus stechende Performance Verlust von Cryptfs lässt sich leicht mit dem Aufwand der Blowfish Verschlüsselung erklären.

Weitere Performance Messungen wurden durch rekursives Datei kopieren und Datei verschieben angestellt. Die Ergebnisse lassen sich mit oben gezeigten vergleichen.

6. Zusammenfassung

Dateisysteme unterliegen im Gegensatz zu Anwendungssoftware einem langsamen Entwicklungsprozess. Obwohl Ideen für nützliche, sinnvolle Erweiterungen vorhanden sind finden diese nur sehr langsam oder überhaupt nicht Einzug in gängige Systeme. Dies liegt vor allem an einer engen Verflechtung von Betriebs- und Dateisystem, die Drittanbietern das Bereitstellen neuer Funktionalitäten extrem erschwert.
Stackable File Systems versuchen dieses Problem zu lösen indem sie eine erweiterbare Architektur schaffen welche es erlaubt einem Dateisystem neue Funktionalität hinzuzufügen, ohne dass es notwendig ist den genauen Aufbau des zu erweiternden Systems zu kennen. Funktionen von verschiedenen Anbietern können beliebig hinzugefügt oder entfernt werden. Dies wird dadurch erreicht, dass jede Schicht ein symmetrisches Interface bietet auf dem neue Schichten aufgesetzt werden können.
Ein weiterer Grund für die langsame Weiterentwicklung ist die Tatsache, dass jede Erweiterung komplexen, systemspezifischen low-Level Code verlangt. Hierfür wurden Sprachen wie FiST entwickelt welche das Programmieren auf einem höheren Abstraktionsniveau erlauben. Der so entstehende Code wird von einem entsprechenden Compiler auf systemspezifischen Maschinencode umgesetzt. Diese Vorgehensweise erlaubt, dass nicht jede neue Dateisystem Funktion, sondern lediglich der Compiler, auf ein neues Betriebsystem portiert werden muss.
Mit FiST können Funktionalitäten die hunderte Zeilen C Code verlangen würden mit nur einigen wenigen Zeilen FiST Code programmiert werden.

Referenzen:

[1] D. S. H. Rosenthal: Evolving the Vnode Interface. USENIX Conference Proceedings, pages 107-118. USENIX, Sommer 1990.

[2] J. S. Pendry and N. Williams: Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha, Imperial College of Science, Technology, and Medicine, London, England, March 1991.

[3] E. Zadock, I. Badulescu, and A. Shender: Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 1988.

[4] E. Zadock, I. Badulescu, and A. Shender: Extending File Systems Using Stackable Templates. USENIX Conf. Proc., 1999

[5] S. R. Kleinman: Vnodes: An architecture for multiple file system types in Sun UNIX. In USENIX Conf. Proc. Juni, USENIX, Berkeley, Calif., 238-247

[6] J. S. Heidemann and G. J. Popek: File System Development with Stackale Layers. ACM Trans.o.Comp.Sys., Vol.12, Nr.1, pages 58-89, Feb., 1994

[7] J. Heidemann and G. Popek: Performance of cache coherence in stackable filing. Fifteenth ACM Symposium on Operating Systems Principles, December 1995.

[8] Ed Skoudis: Counter Hack – A Step-by-Step Guide to Computer Attacks and Effective Defense, ISBN: 0-13-033273-9, page 438

[9] E. Zadok and J. Nieh: FiST: Alanguage for stackable file systems. Usenix Annual Technical Conference, pages 55-70, 2000