SZÉCHENYI ISTVÁN EGYETEM
MŰSZAKI TUDOMÁNYI KAR
JEDLIK ÁNYOS GÉPÉSZ, INFORMATIKAI ÉS VILLAMOSMÉRNÖKI INTÉZET
MATEMATIKA ÉS SZÁMÍTÁSTUDOMÁNY TANSZÉK

MŰSZAKI INFORMATIKA SZAK
szoftverfejlesztés szakirány

DIPLOMAMUNKA

A NETFILTER/IPTABLES bemutatása egy beépülő modul fejlesztésén keresztül

Gáspár Lajos
mérnök-informatikus

2008

Feladat:

A Linux NETFILTER/IPTABLES páros bemutatása.

Részfeladatok:

a) A szakdolgozat elkészítéséhez szükséges résztevékenységek:

A tűzfalrendszerek és különbségeik megismerése. A Linux kernel tűzfalfunkciókat biztosító részének megismerése. Modultervezés. Programozás. Tesztelés.

b) A szakdolgozat főbb részei:

A tűzfalak fogalma. A NETFILTER és az IPTABLES alrendszereinek bemutatása. Modulkészítés lépései, részegységek működésének bemutatása. Dokumentáció. Tesztrendszer. Összefoglalás.

Belső konzulens: Kallós Gábor, SZE JÁI Matematika és Számítástud. Tanszék, Győr
Külső konzulens: Kadlecsik József, főosztályvezető KFKI-SzHK, Budapest

A diplomamunka benyújtásának határideje: 2008. május 2.

Győr, 2008. április 30.

.....	.....	.....
tanszékvezető	Szakfelelős	szakirány-felelős

A diplomamunkát ellenőriztem: ***beadható*** **–** ***nem adható be***

_____ _____

Dátum belső konzulens aláírása

A diplomamunkát ellenőriztem: ***beadható*** **–** ***nem adható be***

_____ _____

Dátum külső konzulens aláírása

Bíráló adatai: .....

.....

.....

A diplomamunka minősítése

A bíráló javaslata:

_____	_____	_____
érdemjegy	dátum	bíráló aláírása

A Tanszék javaslata:

_____	_____	_____
érdemjegy	Dátum	tanszékvezető aláírása

Az ÁVB határozata:

_____	_____	_____
érdemjegy	Dátum	ÁVB elnök aláírása

GÁSPÁR LAJOS

A NETFILTER/IPTABLES BEMUTATÁSA EGY BEÉPÜLŐ MODUL FEJLESZTÉSÉN KERESZTÜL

ÖSSZEFOGLALÁS

Napjainkban az Internet használata már mindennapos eseménynek számít. Azonban a használat során a velünk kapcsolatba lépő számítógépek nem mindegyike jó szándékú. Ezért védelmi lépéseket kell tennünk.

A hálózatok közti védelmi funkciók ellátására részben tűzfalakat használunk. Természetesen nem minden feladat megoldására lehet ilyen úgynevezett határvédelmi eszközöket használni. A tűzfal fő funkciója a védendő hálózat biztonságpolitikájának rákényszerítése a rajta áthaladó kapcsolatokra, adatokra.

A legtöbb ma használatos eszközben Linux operációs rendszeren futó hálózatvédelmi szoftvereket találunk. Ezek működési elve az idők során folyamatosan változott, lépést tartva az újabb betörési technikákkal. Jelenleg a negyedik generáció teljesít szolgálatot a Linux kernelben.

A csomagszűrő az operációs rendszer egyik legösszetettebb alrendszere. Bonyolultsága köszönhető annak is, hogy könnyű új funkciókkal kiegészíteni. Sok olyan bővítmény készült már el hozzá, amelyekkel „törhetetlen” védelmi eszközöket építhetünk.

Diplomamunkámmal a Linux kernel csomagszűrő funkciókat ellátó rendszerét, a NETFILTER/IPTABLES párost mutatom be egy új modul fejlesztésének lépésein végighaladva.

Kulcsszavak: Internet, NETFILTER, IPTABLES, tűzfal, Linux

LAJOS GÁSPÁR

INTRODUCING THE NETFILTER/IPTABLES SYSTEM THROUGH BUILDING A PLUGIN

SUMMARY

Nowadays, the use of Internet is to be considered as an everyday task. However, not every computer we are connected to is a good willing one. For that reason, some steps must be taken in order to protect ourselves.

Partly firewalls are used to provide some protection functions between connected networks. Of course, these so called border-guard equipment can not be applied for every kind of protection. The main function of firewalls is to force the protected network's security-policy on the passing through connections and data.

Most of the today's network protecting equipment runs under the Linux operating system. The operating principle has been continuously changed through the time to keep up with the intrusion techniques. At the moment the fourth generation is on duty in the Linux kernel.

The packet filtering subsystem is considered to be one of the most complicated subsystems of the operating system. Its complexity is due to the fact that it is easy to be extended with new functions. Many new modules exist that helps us to create "unbreakable" defense equipment.

In my thesis the NETFILTER/IPTABLES packet filtering subsystem of the Linux kernel is to be introduced through the steps of the creation of a new module.

Keywords: Internet, NETFILTER, IPTABLES, firewall, Linux

TARTALOMJEGYZÉK

1. Bevezetés 3
2. A Linux 4
2.1. Miért éppen a Linux? 4
3. A tűzfal 5
3.1. Hogyan csoportosítjuk a tűzfalakat? 5
3.1.1. Csomagszűrő tűzfalak 6
3.1.2. Áramkör szintű átjárók 6
3.1.3. Alkalmazás szintű átjárók 7
3.1.4. Állapottartó csomagszűrő tűzfalak 7
4. A NETFILTER 9
4.1. A NETFILTER alrendszerei 9
4.1.1. raw alrendszer 9
4.1.2. conntrack alrendszer 10
4.1.3. mangle alrendszer 10
4.1.4. nat alrendszer 10
4.1.5. filter alrendszer 11
4.1.6. Csomagátadás a felhasználói tér programjainak 11
5. Az IPTABLES 12
5.1. Egyezőségek 12
5.2. Célpontok 13
5.3. Szabályok 14
5.4. Irányelvek 14
5.5. Láncok 15
5.5.1. Saját láncaink 15
5.5.2. PREROUTING lánc 16
5.5.3. INPUT lánc 16
5.5.4. FORWARD lánc 16
5.5.5. OUTPUT lánc 17
5.5.6. POSTROUTING lánc 17
5.6. Táblák 17
6. Modulépítés 18

6.1. Első lépések 18
6.1.1. Forráskódok beszerzése 18
6.1.2. Kernelkonfigurálás 18
6.1.3. Fordítás, tesztelés 19
6.2. Modulkészítés 20
6.2.1. Közös fejléc fájl 21
6.2.2. Kernel rész 22
6.2.3. Felhasználói rész 27
6.3. Fordítás, telepítés, tesztelés 33
7. Összegzés 34
8. Melléklet 35
8.1. Ábrák 35
8.2. Forráskód 38
8.2.1. Közös fejléc fájl 38
8.2.2. Kernel rész 39
8.2.3. Felhasználói rész 41
9. Irodalomjegyzék 45

1. BEVEZETÉS

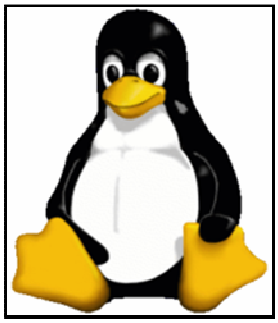
Napjainkban az Internet egyre nagyobb teret hódít az emberek életében. Ha például információt keresünk, vagy rég nem látott ismerősökkel szeretnénk megosztani a velünk történt eseményeket, akkor gyorsan és egyszerűen megtehetjük ezt, köszönhetően a ma már szinte mindenki számára elérhető Világhálónak.
Azonban, ha azt is figyelembe vesszük, hogy veszélyek leselkednek ránk a szabad információáramlást biztosító hálózaton, akkor könnyen beláthatjuk, hogy meg is kell védenünk magunkat. Ha nem csak a saját gépünk sorsa forog kockán, akkor komolyan meg kell ismernünk a veszélyeket, és a védelem lehetőségeit.
A veszélyek sokfélék: vírusok, trójaiak, kémprogramok, kéretlen levelek (*spam*) okozhatnak károkat, terhelhetik meg informatikai rendszereinket. Ugyanakkor nem szabad megfeledkeznünk a digitális kártevők mellett az emberi tényezőről sem. Hiába alakítunk ki bármilyen biztonságosnak hitt rendszert, ha a felhasználók nem tartják be a szabályokat, vagy akár akaratukon kívül olyan információkat szolgáltatnak ki, amelyekkel hozzáértő támadók könnyedén hozzáférhetnek fontos információinkhoz.
A kár, ami érhet bennünket, lehet akár egy e-mail elvesztése, de akár teljes rendszerek összeomlása is.
Védekezhetünk ezek ellen a támadások ellen, és fel is készülhetünk rájuk megfelelő megelőző lépések megtételével. Alkalmazhatunk központi vírusvédelmi rendszereket, üzenetszűrőket, tűzfalakat.

A védelem és egyben a megelőzés egyik, de nem kizárólagos hasznos eszköze lehet egy Linux alapú tűzfal.

A tűzfalak működésének megismerése során sokféle problémával és azok megoldásával találkoztam. Ezekhez olyan ismeretekre volt szükségem, amelyeket a rendszerek működésének részletes tanulmányozása nélkül nem tudtam volna elsajátítani.
A tűzfalak mindegyike alapvetően a csomaggal, kapcsolattal, vagy ezek állapotával foglalkozik. A célkitűzésem az volt, hogy ezt a rendszert kiegészítsem egy olyan modullal, ami a tűzfal hálózati interfészének állapota, paraméterei alapján segít döntést hozni.

Diplomamunkám a NETFILTER/IPTABLES párost mutatja be egy beépülő modul fejlesztésén keresztül. A fejlesztés lépései során megismertetem az olvasót a rendszer belső működésével, ezen belül a kernel és a felhasználói programok közötti kommunikációval. Bemutatom a rendszer vázát adó programozási struktúrákat, szabályokat. Röviden vázolom a fontosabb parancsokat, felhasználási lehetőségüket.
Ugyanakkor nem szándékozom teljes mélységében bemutatni a rendszer alkalmazási lehetőségeit, mivel sokféle tűzfal építhető, és ezek részletes tárgyalása nagyobb terjedelmet, és nem utolsó sorban konkrét megoldandó feladatokat igényel.

2. A LINUX



Linus Benedict Torvalds 1991. augusztus 25-én a *„comp.os.minix”* levelezőlistán jelentette be az általa fejlesztett rendszermagot. [5] Gyakorlatilag ekkor született meg az a szabad forráskódú UNIX változat, amely komoly fejlesztői és felhasználói tábort tudhat maga mögött. Az önkéntes fejlesztők folyamatosan dolgoznak azon, hogy szinte az összes ma ismert architektúra alatt futtatható legyen a Linux. Sok helyen találkozhatunk olyan számítógépekkel, mobiltelefonokkal, beágyazott eszközökkel, amelyek központi vezérlőegységén ez az operációs rendszer fut. Előfordul, hogy egyszerű PC-kből építenek olyan fürtöket, melyek teljesítménye a nagyobb szuperszámítógépekkel vetekedik.

Nemegyszer tűzfalakat hoznak létre vele, hogy akár nagyobb és sérülékenyebb rendszerek védelmét bízzák rá arra a rendszerre, amely anno hobbiból, kíváncsiságból jött létre. Sikerét annak is köszönheti, hogy forráskódja mindenki számára szabadon hozzáférhető, és ezért az újabb eszközök, technológiák támogatása könnyen megvalósítható benne. Ez a fajta flexibilitás teszi lehetővé, hogy az igényeink szerinti rendszert tudjunk a Linux rendszermag köré építeni.

2.1. Miért éppen a Linux?

Erre a kérdésre válaszként az operációs rendszer sok jó tulajdonságát felsorolhatjuk. Ezek között lehet az ingyenesség, a szabad forráskód, a megbízhatóság, és még sok olyan fogalom, ami más operációs rendszerekre csak részben vagy egyáltalán nem igazak.

Az én választásom azért esett a Linux-ra, mert:

- hálózatkezelése etalonnak tekinthető,
- könnyen felépíthető vele egy igényekre szabott tűzfalrendszer,
- jól dokumentált a programozási felülete.

A Linux tűzfal magja a NETFILTER/IPTABLES páros.

1. táblázat Operációs rendszerek és tűzfalak

	Microsoft Windows	Debian Linux
Kernelbe épített tűzfal	✓	✓
Szabad forráskód	✗	✓
Moduláris tűzfal	✗	✓

3. A TŰZFAL

Mielőtt megismerkedünk a tűzfalakkal, fontos, hogy megfogalmazzuk, hogy mit értünk az elnevezés alatt. Az alábbi felsorolás pontokba szedve írja le az informatikai tűzfal fogalmát:

- Olyan hálózati eszközök alkotta egység, amely segítségével kapcsolat hozható létre más, esetenként különböző működésű hálózatok között. Ebben a megfogalmazásban az eszközök hálózati topológiája a mérvadó. Fontos megjegyezni, hogy ebben az egységben azt a kapcsolódási eszközt is a tűzfal részének kell tekinteni, amely adott esetben csak kapcsolatot biztosít, de védelmet nem.
- Szabályok rendszere, amelyek pontosan megfogalmazzák a hálózatok közötti átjárhatóság rendjét, mikéntjét. Itt különbséget tehetünk az alapján, hogy alapértelmezetten a stratégiánk a tiltás vagy az engedélyezés. (Minden tilos kivéve, amit engedünk, vagy ellenkezőleg, minden engedélyezett kivéve, ami tiltott.) Ebben az értelmezésben nem foglalkozunk a kapcsolatok tartalmával, hanem csak az irányokat, mint kapcsolódási lehetőségeket értelmezzük.
- Védelmi lépések, amelyek az általunk védeni próbált eszközök sérülékenységét próbálják elfedni a potenciális támadások elől. Itt már az is számít, hogy milyen adatok közlekednek a kapcsolat résztvevői között.

Egy igazi tűzfal az előbbi állításoknak egyszerre tesz eleget, hiszen szabályokat csak akkor tudunk érvényesíteni két vagy több hálózat között, ha azok forgalma a tűzfalon keresztül bonyolódik, és a tűzfal dönt arról, hogy egy adott kapcsolat megengedhető-e vagy sem.

Röviden összefoglalva a tűzfal a hálózatok között lévő olyan védelmi eszköz, amely rákényszeríti az átmenő forgalomra az előzetesen meghatározott biztonsági szabályokat.

Fontos, hogy ne azonosítsuk a tűzfalat úgy, mint egy csak szoftveres, vagy csak hardveres megoldás, inkább fogjuk fel az optimális hálózati biztonság elérésének egyik eszközeként. Természetesen ez az eszköz is csak addig alkalmas feladatainak ellátására, amíg a védendő hálózatban is alkalmazunk szabályokat, és azt nem csak a külső fenyegetéséktől próbáljuk megvédeni.

3.1. Hogyan csoportosítjuk a tűzfalakat?

A tűzfalakat 4 fő osztályba soroljuk működési elvük alapján:

- Csomagszűrő tűzfal (*Packet filter firewall*)
- Áramkör szintű átjáró (*Circuit-level gateway*)
- Alkalmazás szintű átjáró (*Application-level gateway*)
- Állapottartó csomagszűrő tűzfal (*Stateful Packet Inspection firewall*)

3.1.1. Csomagszűrő tűzfalak

Ebbe a csoportba tartozó tűzfalak nem figyelik a kapcsolatban közlekedő adatok tartalmát, értelmét, hanem csak a csomag paramétereit elemzik. Működésük az ISO-OSI modellben (1. ábra. *Az ábrák a 8.1-es fejezetben találhatók.*) a hálózati és a szállítási rétegre korlátozódik. Ezért alig többek egy *router*-nél.

Egy csomag sorsa a fejlécében tárolt információk alapján dől el:

Hálózati rétegben (IP (2. ábra) /ICMP/IGMP/ARP/RARP fejléc alapján):

- Honnan jött?
- Hova megy?
- Milyen protokoll?
- Milyen hosszú a csomag?

Szállítási rétegben (TCP (3. ábra) /UDP fejléc alapján):

- Mi a forrás cím?
- Mi a forrás port?
- Mi a cél cím?
- Mi a cél port?
- Milyen típusú csomag a protokollon belül?

Ha a címfordítási funkcióktól eltekintünk, akkor elmondhatjuk, hogy az ebbe a csoportba tartozó tűzfalak alapvetően nem módosítanak csomagtartalmat.

Előnyök:

- Olcsó eszközökben is találkozhatunk már vele.
- Transzparens a felhasználók számára.
- Gyorsan eldől a csomagról, hogy továbbítható-e.

Hátrányok:

- Hosszú, bonyolult szabályokkal sem érünk el optimális biztonságot.
- A döntések nem a teljes kapcsolat alapján dőlnek el.
- Azonosítás nincs ezen a szinten.

3.1.2. Áramkör szintű átjárók

Ezek a tűzfalak azt döntik el, hogy megengedhető-e a kapcsolat annak függvényében, hogy a résztvevők betartják-e a kapcsolat-felépítés szabályait, helyesen használják-e a jelzőbiteket. Ilyen feladatokra a csomagszűrő tűzfal nem képes, mivel nem ismeri a kapcsolat állapotát.
A kliens nem közvetlenül a kiszolgálóval beszél, hanem megkéri az átjárót a nevében való eljárásra. Ha felépült a kapcsolat, akkor az átjáró már csak továbbítja a csomagokat mindkét irányba. Ha valamelyik fél bontja a kapcsolatot, akkor a másik kapcsolat is véget ér. Ezek a funkciók az ISO-OSI modell viszonylati rétegében valósulnak meg.
Gyakorlatilag ezek a tűzfalak a *proxy*-k.

(Fontos, hogy ne keverjük össze a *proxy*-kat a *webcache*-ekkel! A *webcache* a *proxy*-val ellentétben nem csak létrehozza a kapcsolatot a két hálózat között, hanem a letöltött adatokat a saját tárában gyorsítótárazza, A későbbi egyező kéréseket már ebből a helyi tárból szolgálja ki, ha a forrás közben nem változott meg. Ezzel csökkenti a hálózat válaszidejét, illetve növeli az áteresztőképességét.)

Előnyök:

- „Áramkör szintű” kapcsolatot hoznak létre két hálózat között.
- Elrejtik a kapcsolat valódi kezdeményezőjét.
- Elrejtik a védendő hálózat felépítését.
- Teljes felügyelet a kapcsolat létrehozása során.
- Azonosítást kérhet a kapcsolat felépítése előtt.

Hátrányok:

- A kapcsolat lassabban épül fel a két végpont között.
- A klienst konfigurálni kell az átjáró használatához.
- Becsapható, mivel nem figyeli a tartalmat.
- Nem a kernelben fut az átjáró szoftvere, ezért lassabb.

3.1.3. Alkalmazás szintű átjárók

Ezek a tűzfalak ugyanúgy nem engednek át közvetlenül magukon kapcsolatokat, mint az áramkör szintű átjárók, azonban már értik is azt a protokollt, amit továbbítanak a két végpont között. Megadható, hogy egy protokollon belül milyen parancsok használhatók. Egyik tipikus felhasználási területük ezeknek az átjáróknak az e-mail tartalomszűrők, amelyek például a csatolt állományokban keresnek vírusokat, férgeket, illetve meg nem engedett tartalmakat. Ezek a tűzfalak, – ahogy a nevük is sugallja – az alkalmazási rétegben működnek.

Előnyök:

- Protokoll szintű szabályok hozhatók létre.
- Elrejti a kliens verzióját, és ez által védi a verziófüggő támadásokkal szemben.

Hátrányok:

- Általában egyszerre több protokollt nem ismer.
- Folyamatosan figyeli az adatfolyamot, és ezért lassít.
- Nem biztos, hogy a protokollbővítéseket támogatja.

3.1.4. Állapottartó csomagszűrő tűzfalak

Ha egy csomagszűrő tűzfalat felvértezünk az áramköri szintű és az alkalmazás szintű átjáró előnyös tulajdonságaival, akkor eljutottunk az állapottartó csomagszűrő tűzfalak csoportjához. Az egyesítéssel lehetőségünk nyílik a kapcsolat felépítését felügyelni, mint az áramkör szintű átjárónál, és kiegészítő modulok felhasználásával protokoll szintű szabályok is alkalmazhatók, mint az alkalmazás szintű átjárónál.

Természetesen lemondásokkal is jár egy ilyen egyesítés. Nincs lehetőségünk proxy funkciók ellátására, felhasználók azonosítására és hozzáférési jogosultságuk ellenőrzésére, ezért ezek megvalósítására kiegészítő szoftvereket kell alkalmaznunk.
Ez a tűzfaltípus az adatkapcsolati rétegtől az alkalmazási rétegig képes szűrni a kapcsolatot.

Előnyök:

- Gyors, mert a kernelben fut.
- Csomagszűrő, és ezért transzparens.
- Teljes kapcsolat alatt ellenőrizheti a szabályok betartását.
- Mivel állapottartó, ezért nem csak a csomag által hordozott információkat, hanem a csomag a kapcsolatban betöltött szerepét is vizsgálja.
- A kiegészítő információkat, kapcsolódó kapcsolatokat is figyelheti.

Hátrányok:

- Nem tud teljes protokoll-szűrést végrehajtani.
- Nem kezdeményez, és tart fent kapcsolatot bármely kliens javára.
- Komoly szakértelem kell a szabályrendszerek felállításához.

4. A NETFILTER

A Linux kernel negyedik generációs tűzfalfunkciókat biztosító keretrendszere a NETFILTER. Az erre épített tűzfalak az állapottartó csomagszűrő tűzfalak körébe tartoznak.

Főbb tulajdonságai:

- Ez a rész az operációs rendszer magjának része, ezért hibamentesnek kell lennie.
- Kernel módban való futása miatt gyorsan dönt az adott csomag sorsáról.
- Felépítését tekintve moduláris, függetlenül attól, hogy a modulokat valóban modulként, vagy a kernelbe fordítva hoztuk-e létre.
- API hívásokon keresztül egy teljes keretrendszert biztosít a szabályok felépítéséhez, újabb modulok rendszerbe integrálásához.
- Az újabb modulokkal egyre bonyolultabb, és pontosabb tűzfal hozható létre.

NETFILTER a neve annak a csoportnak is, akik a Linux kernel tűzfal részével foglalkoznak.

4.1. A NETFILTER alrendszerei

A NETFILTER-ben öt alrendszert találunk, amelyek különböző feladatokat végeznek:

- *raw* = alrendszerek szabályozása,
- *conntrack* = kapcsolatkövetés,
- *mangle* = csomagmódosítás,
- *nat* = címfordítás,
- *filter* = csomagszűrés.

Egy csomag csak akkor jut át a tűzfalon, ha minden alrendszeren áthaladt.

4.1.1. raw alrendszer

Ez a legkisebb és egyben a legújabb alrendszer. Itt lép be először a csomag a NETFILTER-be. Jelenleg két feladatot lát el:

- Megadhatjuk az utána következő *conntrack* alrendszernek, hogy mely kapcsolatokat nem kell figyelnie. Ilyen kapcsolatok lehetnek például a helyi gépről helyi gépre irányuló, úgynevezett *loopback* kapcsolatok.
- A rendszeren áthaladó csomagokról az útvonalukat leíró naplóbejegyzést készíttethetünk. Ezzel lehetőségünk nyílik arra, hogy a szabályrendszerünket ellenőrizzük, javítsuk.

4.1.2. conntrack alrendszer

Ez az alrendszer biztosítja, hogy állapottartó legyen a tűzfal. Belső állapotterével megpróbálja elérni, hogy megmondható legyen egy csomagról, hogy az a kapcsolat melyik állapotához tartozik. Mivel állapottere hasonlít a TCP állapotteréhez, ezért gyakran összekeverik a két rendszert, ami félreértéseket okozhat.

Az alrendszer a következő állapottérben értelmezi a csomagokat a kapcsolaton belül:

- *NEW* = A csomag egy új kapcsolatot hozott létre, vagy a rendszer által még nem ismert, de valós kapcsolat része.
- *ESTABLISHED* = A csomag olyan kapcsolathoz tartozik, amelyben már mindkét irányban volt adatforgalom.
- *RELATED* = A csomag egy új kapcsolatot hozott létre, de egy már létező kapcsolathoz tartozik, annak kiegészítése. Ilyen lehet például az FTP adatkapcsolati csatornája.
- *INVALID* = A csomag nem tartozik hozzá egyetlen ismert kapcsolathoz sem.
- *SNAT* = Virtuális állapot, amely arról tájékoztat, hogy a csomag egy forráscím címfordításon ment keresztül.
- *DNAT* = Hasonlóan az SNAT-hoz, itt is címfordítás történt, de itt a célcím változott meg.
- *UNTRACKED* = Ha a raw alrendszerben megadjuk egy kapcsolatról, hogy figyelmen kívül hagyandó, akkor ezt az állapotot veszi fel a conntrackban.

4.1.3. mangle alrendszer

A mangle alrendszerben van lehetőségünk a csomag tartalmának, tulajdonságainak megváltoztatására.
Ilyen tulajdonságok lehetnek például az ECN bitek[1] állításai, vagy akár a maximális szegmens méret információ beállítása. Lehetőség van a kapcsolat illetve csomag megjelölésére is, amit később a rendszer akár az útvonalválasztáshoz is felhasználhat.

4.1.4. nat alrendszer

A nat[2] alrendszer segítségével tudunk címfordítást elérni.
Tipikus felhasználási terület, amikor a védendő hálózat részére végzünk címfordítást. Ilyenkor a belső címekről érkező kérést úgy továbbítja a tűzfal a külső hálózat – adott esetben az Internet – felé, hogy a kimenő csomagokban a küldő címét átírja a saját külső címére. Ekkor a címzett úgy érzékeli, mintha a tűzfalról származnának a kérések, és ezért neki válaszol. A visszaérkező válasz csomagoknál ennek az ellenkezőjét teszi, azaz visszacseréli a célállomás címét (saját külső címét) a belső eredeti kérő címére, és a csomagot a belső hálózatra továbbítja.

[1] ECN (Explicit Congestion Notification) = Nagysebességű hálózatokhoz kifejlesztett csomagtorlódást jelző rendszer. [1]
[2] NAT (Network Address Translation) = Hálózati címfordítás.

Bár a csomag címei is olyan adatok, melyeket meg lehetne változtatni a mangle alrendszerben, mégis a rendszer a címfordítást külön kezeli. Ez azért is történik így, mert a kapcsolat idején a címfordítás automatikusan mindkét irányban végbemegy, és ehhez egy olyan teljesen független alrendszer kell, amely megjegyzi az ilyen kapcsolatokat.
A címfordítás lehet forrás vagy cél cím átírás, illetve átirányítás.

4.1.5. filter alrendszer

Ebben az alrendszerben tudunk dönteni arról, hogy továbbengedjük-e a csomagot, vagy sem. A valódi csomagszűrést itt érdemes megvalósítani.
Bár az előző alrendszerekben is eldobhatunk csomagot, ez mégsem ajánlott, mert így összekeverjük az alrendszerek feladatait, és egy későbbi módosítás során nem várt eredményt produkálhat a tűzfalunk.

4.1.6. Csomagátadás a felhasználói tér programjainak

A NETFILTER egy érdekes lehetőséget is biztosít az olyan feladatok megoldására, amelyek felépítésével, alrendszereivel nem megoldhatók. Lehetőségünk van egy aszinkron soron keresztül, a felhasználói térben futó folyamat számára átadni csomagokat, hogy az döntse el, mi legyen a kapcsolat további sorsa. Ugyan ez nem egy külön alrendszer, mégis működését tekintve élesen elhatárolódik a rendszer többi részétől.
Ha nincs regisztrálva egyetlen felhasználói program sem az adott csomagtípusra, akkor a csomagot a rendszer eldobja.
Nincs megkötés abból a szempontból, hogy mit csinálhat a program a csomagokkal. Kicserélhet adatokat, eltávolíthat érzékeny információkat, vagy akár kódolhat/dekódolhat is csomagokat. Feladata befejeztével a NETFILTER API hívásain keresztül visszaküldheti a kernelbe a csomagot további feldolgozásra, továbbításra.

Általánosságban elmondható, hogy csak nagyon speciális esetekben van erre a funkcióra szükség. Tipikus eset, amikor úgynevezett IDS/IPS rendszerek számára adunk át csomagokat. (*Intrusion Detection System – Behatolás-érzékelő rendszer; Intrusion Prevention System – Behatolás-megelőző rendszer*) Ezek a rendszerek a kapcsolatokat nem csak adatfolyamokként, hanem eseményekként is kezelik, és összefüggéseket keresnek köztük. Így tehát lehetőséget kínálnak arra, hogy felismerhessük a potenciálisan veszélyes forrásokat, és adott esetben megelőző lépéseket tegyünk a védelem érdekében.

5. AZ IPTABLES

Mivel a csomag sorsa valójában a kernelben dől el, ezért kell egy kommunikációs felület, amin keresztül elérjük a kernelt. Ezért fogalmazhatunk úgy, hogy az IPTABLES a NETFILTER *ipv4*-re fejlesztett felhasználói felülete. Természetesen létezik *ipv6*-ra *arp*-re és *bridge* rendszerekre/funkciókra is ilyen felület, sorra *ip6tables*, *arptables* és *ebtables* néven.
Az *iptables* paranccsal tudjuk feltölteni/törölni a szabályainkat, illetve lekérdezni a rendszer aktuális állapotát.

A rendszerbe kerülő csomag az IPTABLES tábláiban (*table*) meghatározott láncain (*chain*) végighaladva, az ott kialakított szabályok (*rule*) alapján kerül feldolgozásra.
A szabályok két részből állnak: egyezőségek (*match*) és célpontok (*target*).

Ha egyetlen (*terminális*) szabály sem illeszkedett a csomagra, akkor a láncra érvényes irányelv (*default policy*) lesz a meghatározó a csomag sorsát illetően.

A rendszert a legkisebb elemektől a legnagyobbakig haladva érdemes megismerni.

5.1. Egyezőségek

Az egyezőségekkel tudunk kiválasztani bizonyos tulajdonságoknak megfelelő csomagokat a tűzfalra érkezőek közül. Gyakorlatilag ezek a feltételrendszerek kritériumai. Kapcsolati állapotokat, csomag fejléceket, vagy akár csomag tartalmakat is vizsgálhatunk a segítségükkel. Míg egyes egyezőségek bármely táblában, addig néhány csak bizonyos táblákban értelmezhető. (Például nincs értelme a helyi gép által küldött csomagnál bejövő interfészről beszélni.)

A beépített egyezőségeken kívül léteznek a külső fejlesztők által készített kiegészítők is. Ezeket az úgynevezett *patch-o-matic-ng* [3] csomagban adják közre. Ha ezekkel a külső modulokkal kiegészítjük a tűzfalunkat, akkor növelhetjük megbízhatóságát, rugalmasságát.

Előfordul, hogy egy ilyen kiegészítőt kell alkalmaznunk, mert csak ennek a segítségével tudjuk szabályozni a két hálózat közötti kommunikációt.

Ilyen esetre lehet példa, amikor a különféle fájlcserélő hálózatok forgalmát szeretnénk tiltani. Alapesetben elég nehéz helyzetbe kerülünk. Ezek a hálózatok nem használnak a protokollra jellemző portcímeket, vagy központosított kiszolgálókat, amik alapján tilthatnánk a kapcsolatot. Ilyenkor segítségünkre lehet az adatforgalom kiszűrésében, ha egy olyan egyezőséget használunk, amely átvizsgálja a csomagok tartalmát, olyan parancsok után kutatva, amelyek jellemzőek az adott protokollra. Ha talál ilyent, akkor egyszerűen eldobhatjuk a kapcsolatot.

Néhány példa az egyezőségekre:

- *-i eth0* = A csomag az eth0 interfészen érkezett.
- *-o ppp0* = A csomag a ppp0 interfészen távozik.
- *-p tcp* = A csomag a TCP szállítási rétegen keresztül közlekedik.
- *-s 192.168.0.1* = A csomag forrása a 192.168.0.1-es IP című gép.
- *-dport 80* = A csomag célja a 80-as port.
- *-m ipp2p --bit* = Bittorrent protokollt alkalmazó átvitel része a csomag.
- *-m owner --owner-uid 0* = A csomag a *root* felhasználó által indított folyamattól származik.

A feltétel érvényességének megfordítására a felkiáltójelet használjuk.
Az egyezőségek neveit kisbetűvel írjuk.

5.2. Célpontok

A célpontok adják meg, hogy mi történjen a csomaggal. A legegyszerűbbek csak a csomag továbbengedhetőségét befolyásolják, de léteznek olyanok is, amelyek megváltoztatják a csomag tartalmát.

Célpont lehet az általunk definiált lánc is. Ebben az esetben a csomag a láncunkban folytatja tovább az útját.

A *célpont* lehet *terminális* vagy *nem terminális*. A *terminális célpont* végrehajtása után a csomag nem folytatja tovább az útját a láncban.

Egységesen elmondható, hogy azok a célpontok, amelyek megváltoztatják, megjelölik vagy naplózzák a csomagot, nem terminálisak.
Viszont azok a célpontok, amelyek elfogadják vagy elutasítják a csomagot, egyértelműen terminálisak. Ez utóbbi csoportnál, amíg az elutasító célpontok teljes egészében eldobják a csomagot a rendszerből, addig az elfogadó célpont csak az aktuális táblából engedi tovább a csomagot.

Természetesen célpontok is készülnek külső fejlesztők által, és ezeket ugyancsak megtaláljuk a *patch-o-matic-ng* csomagban. Az egyik ilyen érdekes fejlesztés a TARPIT[3] célpont, amely azzal segíthet sokat a kéretlen kapcsolatok számának csökkentésében, hogy használata során a támadót nem hagyja érvényesülni. (Működése ahhoz hasonlítható, mint amikor szeretnénk beszélni valakivel, de az illető folyamatosan azt jelzi, hogy pillanatnyilag nem ér rá, de várjunk türelmesen.) Egy ilyen csapdába ragadt kapcsolat 12-24 percig is várakoztatható, amíg végül a kezdeményező fél lezárja a csatornát.

[3] Szurokcsapda. A küldési ablak 0 méretűre való állításával a küldő rendszerét terheli le. [2]

Néhány célpont:

- *ACCEPT* = A csomagot elfogadjuk a táblában. Terminális.
- *DROP* = A csomagot eldobjuk, mintha nem is létezett volna. Terminális.
- *REJECT* = Ugyancsak eldobjuk a csomagot, de megpróbáljuk értesíteni erről a küldőt. (*TCP RESET*-tel, vagy *ICMP error*-ral.) Terminális.
- *RETURN* = A csomag az aktuális láncból visszakerül a láncot meghívó láncba. Nem terminális.
- *LOG* = A csomagról egy bejegyzés készül a rendszernaplóban. Nem terminális.
- *CONNMARK* = Megjelöljük egy azonosítóval azt a kapcsolatot, amelyiknek része a csomag. Nem terminális.
- *TTL* = A csomag élettartalmát jelző értéket lehet vele változtatni. Nem terminális.
- *OUEUE* = A csomag átadása felhasználói térben futó programnak. Terminális.
- *DNAT* = Célcím átírás. Terminális.

A célpontok neveit NAGYBETŰVEL írjuk.

5.3. Szabályok

A szabályok egy vagy több egyezőségből és egy célpontból állnak. A szabályban lévő célpont alapján a szabályt is nevezhetjük terminálisnak vagy nem terminálisnak.

Ha egy adott kapcsolatra, csomagtartalomra vagy ezek bármely adatára (a továbbiakban csomagra) egyezik a szabályban lévő összes *egyezőség*, akkor a szabályhoz tartozó *célpont* kerül végrehajtásra.

```
iptables -t filter -A INPUT -j ACCEPT -p tcp -s 192.168.0.1
```

Az előbbi példában elfogadjuk (célpont: *ACCEPT*) azokat a csomagokat, amelyek TCP protokollon keresztül érkeznek (egyezőség: *-p tcp*) ÉS a 192.168.0.1-es gép a csomagok küldője (egyezőség: *-s 192.168.0.1*).

5.4. Irányelvek

Az irányelvek speciális szabályok. Ha a láncban a csomagra nincs egyező terminális szabály, akkor a lánc végén ez a célpont lesz végrehajtva. Csak az *ACCEPT* és a *DROP* célpont lehet irányelv. Érdemes az *filter* tábla kivételével az összes táblában *ACCEPT*-re állítani az irányelvet, hogy a rendszer továbbengedje a csomagokat a *filter* tábláig, ahol végül valóban elvégezzük a csomagszűrést.

5.5. Láncok

A NETFILTER a kernel hálózati csomagkezelő rendszeréből öt helyen vesz át csomagokat feldolgozásra. Ezek a kapcsolódási pontok az útvonalválasztással szorosan összefüggnek, amire elnevezésük is utal: (4. ábra)

- *PREROUTING* = Az útvonalválasztás előtti csomagok.
- *INPUT* = A tűzfalnak küldött csomagok.
- *FORWARD* = A tűzfalon áthaladó csomagok.
- *OUTPUT* = A tűzfal által generált csomagok.
- *POSTROUTING* = Az útvonalválasztás utáni csomagok.

Az IPTABLES-ben ezek az elnevezések a beépített láncok nevei. Ezekben a láncokban azokat a szabályokat helyezzük el, amiket a kernelből átvett csomagokon szeretnénk értelmezni.
A láncok több táblában is megtalálhatók, attól függően, hogy az adott tábla mely alrendszerek működését támogatja.
Például csomagszűrést (*filter* tábla) „szabályosan” csak az INPUT, OUTPUT és a FORWARD táblában végezhetünk. Ugyanakkor címátírást (*nat* tábla) csak a PREROUTING, OUTPUT és a POSTROUTING lánc enged meg. Csomagmódosítást pedig bármelyik láncban elvégezhetünk.
Egyes egyezőségek illetve célpontok csak megadott tábla-lánc kombinációban használhatók. Erre példa a *DNAT* célpont, amely csak a *nat* tábla *PREROUTING* és *OUTPUT* láncaiban érvényes.

A saját láncaink kivételével a láncok neveit NAGYBETŰKKEL írjuk.

5.5.1. Saját láncaink

Természetesen saját láncokkal is kiegészíthetjük a táblákat. Ezekkel gyorsíthatjuk és ésszerűsíthetjük a rendszerünket, hiszen ha például különválasztjuk a TCP csomagokra érvényes szabályokat, akkor nem kell ezeken a szabályokon leellenőriznünk az ICMP csomagokat.

A saját láncok csak abban a táblában érvényesek, ahol létrehozzuk őket.

Egy új lánc könnyen felvehető az alábbi paranccsal:

```
iptables -t filter -N bejovo_tcp_forgalom
```

(A fenti parancsnál megadtuk, hogy a *filter* táblában hozzuk létre az új láncot. Ez elhagyható, hiszen ez a tábla az alapértelmezett.)

Ha már van egy általunk definiált lánc, akkor ebbe a láncba illeszthetünk szabályokat, illetve hivatkozhatunk rá más láncokból:

```
iptables -A bejovo_tcp_forgalom -j DROP -s 10.0.0.1
iptables –t filter -A INPUT -j bejovo_tcp_forgalom -p tcp
```

5.5.2. PREROUTING lánc

Amikor a tűzfal egy hálózati kártyán keresztül megkap egy csomagot, akkor el kell döntenie, hogy az neki szól, avagy továbbítandó. A PREROUTING láncban az útválasztáson még át nem esett csomagokat találjuk. Innen az INPUT vagy a FORWARD láncba kerülnek. Ez a lánc főként címfordításnál, átírásnál és kapcsolat nyomkövetésnél kerül előtérbe.

```
iptables -t nat -A PREROUTING –j DNAT -i eth1 -p tcp -dport 5525 --to-destination 192.168.0.100:25
```

(Az *eth1* interfészen az *5525*-ös portra érkező TCP csomagokat irányítsuk át a *192.168.0.100*-as gép *25*-ös portjára.)

5.5.3. INPUT lánc

Ha az útválasztás során kiderül egy csomagról, hogy azt a tűzfalnak címezték, akkor az, az INPUT láncba kerül. A lánc végén egy helyi folyamat kapja meg a csomagot. Ebben a láncban már nem lehet címmanipulációt végrehajtani, illetve nem lehet kimenő interfészre egyezőséget nézni.

```
iptables -t filter -A INPUT –j REJECT –i eth0 ! –s 192.168.0.0/24
```

(Az *eth0* interfészen hibaüzenettel utasítsunk el minden olyan csomagot, ami nem a *192.168.0.0/24*-es alhálózatról érkezett.)

5.5.4. FORWARD lánc

Ezt a láncot a továbbítandó csomagok használják. Fontos hogy észrevegyük, hogy ebben a láncban a csomagok a hálózatok között mindkét irányba közlekednek. Ez után a lánc után a csomagok a POSTROUTING láncban folytatják útjukat. Hasonlóan az INPUT lánchoz, itt sem lehet címeket átírni.

```
iptables -t filter -A FORWARD -j ACCEPT -i eth0 -s 192.168.0.0/24 -o eth1 -d 192.168.1.0/24
iptables -t filter -A FORWARD -j ACCEPT -i eth1 -s 192.168.1.0/24 -o eth0 -d 192.168.0.0/24
iptables -t filter -A FORWARD -j REJECT
```

(Azokat a kapcsolatokat engedélyezzük, amelyeknél az *eth0* interfészen keresztül a *192.168.0.0/24*-es hálózat gépei kommunikálnak az *eth1* interfész *192.168.1.0/24*-es hálózatának gépeivel.)

5.5.5. OUTPUT lánc

A helyi folyamatok által generált csomagok itt kerülnek be a NETFILTER-be. Innen egy újabb útválasztási döntés után távozhatnak a POSTROUTING láncon át, vagy egy másik helyi folyamat bemeneteként fejezik be útjukat az INPUT lánc végén.
Érdekes megfigyelni, hogy az úgynevezett *loopback* kapcsolatok csomagjai először itt jelennek meg. Ezért ha ilyen kapcsolatokon szeretnénk tűzfalszabályokat alkalmazni, akkor azt ebben a láncban érdemes megtennünk. (Mivel azonban a loopback interfész azzal a speciális tulajdonsággal rendelkezik, hogy csak a helyi gép küldhet rá és fogadhat róla csomagokat, ezért általában ilyen szabályokat nem szoktak alkalmazni.)
Ebben a láncban a NETFILTER összes alrendszerét használhatjuk. Hasonlóan az INPUT lánchoz, itt nincs értelme bejövő interfészre egyezőséget nézni.

```
iptables -t filter -A OUTPUT -j ACCEPT -o lo
```

(Fogadjuk el a *loopback* interfészen távozó csomagokat.)

5.5.6. POSTROUTING lánc

Ez az utolsó lánc, mielőtt a csomag egy hálózati kártyán elhagyja a tűzfalat. Fő funkciója a 4.1.4-es fejezetben említett hálózati címfordítás.

```
iptables -t nat -A POSTROUTING -j SNAT -o ppp0 --to-source 193.224.128.3
```

(Minden a *ppp0* interfészen távozó csomagnak írjuk át a forráscímét a *193.224.128.3*-as címre.)

5.6. Táblák

Az IPTABLES táblák megfelelnek a NETFILTER alrendszereinek. Az összes tábla előre definiált, a kernelbe betöltött NETFILTER moduloktól függ. Saját táblákat nem tudunk létrehozni. Az IPTABLES-ben nincs *conntrack* tábla, mivel a *conntrack* alrendszer teljesen önszabályzó. Mivel az IPTABLES táblái a NETFILTER alrendszereinek módosítását, megjelenítését végezik, ezért külön tárgyalni ezt a rendszert nem érdemes. Az alábbi táblázat összefoglalja a táblák és a láncok összefüggéseit.
A táblák neveit kisbetűvel írjuk.

2. táblázat Táblák és láncok.

Lánc /Tábla	PREROUTING	INPUT	FORWARD	OUTPUT	POSTROUTING
raw	✓	✗	✗	✓	✗
mangle	✓	✓	✓	✓	✓
nat	✓	✗	✗	✓	✓
filter	✗	✓	✓	✓	✗

6. MODULÉPÍTÉS

Diplomamunkámmal arra vállalkoztam, hogy bemutatom a NETFILTER/IPTABLES páros belső működését. Létrehoztam egy olyan eddig még nem létező egyezőség modult, amivel lehetőség nyílik olyan szabályokat megalkotni, amelyek figyelembe veszik a tűzfal interfészének állapotát.
A továbbiakban a kivitelezés lépésein vezetem végig az olvasót. Ez a rész főként C nyelvben való programozásról szól. A fejlesztést *Debian* disztribúció alatt végeztem.

6.1. Első lépések

Csak akkor tudunk modult illeszteni a rendszerünkhöz, ha a kernel is képes lesz annak betöltésére, futtatására. Ehhez új kernelt kell fordítanunk, amihez szükségünk lehet a forráskódokra.

6.1.1. Forráskódok beszerzése

Első lépésként szerezzük be a kernel forrását.
Legegyszerűbben ezt a www.kernel.org[4] címről tölthetjük le. Ajánlott a „*stable*” verzió használata. A dokumentum írása pillanatában a 2.6.24.4-es verzió volt elérhető.

A kernel forrásán kívül még szükségünk lesz az IPTABLES forrására is.
Ez a www.netfilter.org[5] honlapról tölthető le.

Szükségünk lesz még a *kernel-package* nevű csomagra is.

Az alábbi parancsok segítséget nyújtanak a fenti lépések végrehajtásához:

```
cd /usr/src
wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.24.4.tar.bz2
wget ftp://ftp.netfilter.org/pub/iptables/iptables-1.4.0.tar.bz2
tar -xjf linux-2.6.24.4.tar.bz2
tar -xjf iptables-1.4.0.tar.bz2
ln –s linux-2.6.24.4 linux
ln –s iptables-1.4.0 iptables
apt-get install kernel-package
```

6.1.2. Kernelkonfigurálás

A letöltött kernel forráskódot ezek után a saját rendszerünkhöz kell igazítani. Mivel általánosan működő beállítások nem léteznek, ezért a következőkben a minimálisan szükséges opciókat adom meg.

[4] http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.24.4.tar.bz2
[5] ftp://ftp.netfilter.org/pub/iptables/iptables-1.4.0.tar.bz2

Kezdjük a konfigurálást a karakteres menüfelülettel:

```
make menuconfig
```

Állítsuk be az alábbi opciókat:

```
CONFIG_NET=y
CONFIG_PACKET=y
CONFIG_NETFILTER=y
CONFIG_NF_CONNTRACK=m
CONFIG_NF_NAT=m
CONFIG_IP_NF_FILTER=m
CONFIG_IP_NF_IPTABLES=m
CONFIG_IP_NF_MANGLE=m
CONFIG_IP_NF_RAW=m
```

További opciók is beállíthatók, attól függően, hogy milyen egyezőségeket illetve célpontokat szeretnénk használni. Javasolt ezeket modulként fordítani, hiszen így a kernel bármikor be tudja tölteni ezeket a kiegészítéseket.

6.1.3. Fordítás, tesztelés

Miután beállítottuk a számítógépünknek megfelelő konfigurációt, lefordíthatjuk a kernelt:

```
cd /usr/src/linux
make-kpkg --bzimage kernel-image
```

Ha hibaüzenet nélkül befejeződött a folyamat, akkor egy *linux-image-2.6.24.4* kezdetű, *.deb* kiterjesztésű fájlt találunk az */usr/src* könyvtárban. Ennek telepítésével elkészültünk a kernelfordítással. Ezek után már csak az IPTABLES fordítása vár ránk:

```
cd /usr/src/iptables
make
make install
```

Teszteléshez első lépésként indítsuk újra a rendszerünket az új kernellel. Ha sikerült belépnünk rendszergazdaként a gépbe, akkor adjuk ki a következő parancsokat:

```
/bin/uname –r
/usr/local/sbin/iptables -V
```

Ezekre rendre az alábbi válaszokat kell kapnunk:

- *2.6.24.4*
- *iptables v1.4.0*

6.2. Modulkészítés

Mielőtt belekezdenénk a programozásba, nézzük meg, milyen állapotokban lehet egy interfész Linuxban. Ehhez a kernel *if.h* fejléc (*header*) fájlját kell átvizsgálnunk. (Ezt az alábbi útvonalon találjuk: */usr/src/linux/include/linux/if.h*)
A fejléc fájlban az alábbi definíciót találjuk:

```
/* Standard interface flags (netdevice->flags). */
#define IFF_UP          0x1             /* interface is up              */
#define IFF_BROADCAST   0x2             /* broadcast address valid      */
#define IFF_DEBUG       0x4             /* turn on debugging            */
#define IFF_LOOPBACK    0x8             /* is a loopback net            */
#define IFF_POINTOPOINT 0x10            /* interface is has p-p link    */
#define IFF_NOTRAILERS  0x20            /* avoid use of trailers        */
#define IFF_RUNNING     0x40            /* interface RFC2863 OPER_UP    */
#define IFF_NOARP       0x80            /* no ARP protocol              */
#define IFF_PROMISC     0x100           /* receive all packets          */
#define IFF_ALLMULTI    0x200           /* receive all multicast packets*/

#define IFF_MASTER      0x400           /* master of a load balancer    */
#define IFF_SLAVE       0x800           /* slave of a load balancer     */

#define IFF_MULTICAST   0x1000          /* Supports multicast           */

#define IFF_PORTSEL     0x2000          /* can set media type           */
#define IFF_AUTOMEDIA   0x4000          /* auto media select active     */
#define IFF_DYNAMIC     0x8000          /* dialup device with changing addresses*/

#define IFF_LOWER_UP    0x10000         /* driver signals L1 up         */
#define IFF_DORMANT     0x20000         /* driver signals dormant       */
```

Az állapotok bővebb leírását az *RFC2863*-ban [4], illetve az *operstates.txt*-ben találjuk (*/usr/src/linux/Documentation/networking/operstates.txt*).

Mivel néhány állapot csak történelmi okokból található meg a felsorolásban, ezért csak az alábbi állapotokat fogjuk használni a modulunkban:

- *IFF_UP* = Az interfész működik. (Adminisztratív szempontból.)
- *IFF_BROADCAST* = Az interfész „szórt” üzeneteket is fogad.
- *IFF_LOOPBACK* = Az interfész *loopback* eszköz.
- *IFF_POINTOPOINT* = Az interfész két pont közti kommunikációt biztosít.
- *IFF_RUNNING* = Működőképes állapot. (Fizikailag.)
- *IFF_NOARP* = Nem figyel ARP csomagokat.
- *IFF_PROMISC* = Minden csomagot elfogad.
- *IFF_MULTICAST* = *Multicast* csomagokat is elfogad.
- *IFF_DYNAMIC* = Az eszköz dinamikusan lett létrehozva.
- *IFF_LOWER_UP* = Az alrendszerek működőképes állapotban vannak.
- *IFF_DORMANT* = Eseményre, kapcsolatra várakozik.

6.2.1. Közös fejléc fájl

A modul legkisebb eleme a fejléc fájl, amiben megadjuk, hogy milyen struktúrákat, konstansokat használnunk közösen a kernelben és a felhasználói tér programjában. Ennek a fájlnak *xt_iface.h* lesz a neve.

```
#define DEBUG 0
#define _MODULE_NAME "iface"
#define _MODULE_REVISION 0
```

Az első három konstans beállítja az alábbi opciókat:

- *DEBUG* = Hibakeresés be illetve kikapcsolása. Nullától eltérő érték esetén a rendszernaplóban és a konzolon is kaphatunk visszajelzéseket arról, hogy mit csinál a modul, illetve hogy mely részei lettek elindítva.
- *_MODULE_NAME* = A modul nevét adja meg. Ezzel az üzenetekben szereplő nevet tudjuk könnyedén megváltoztatni.
- *_MODUL_REVISION* = A modul revíziószámát állíthatjuk be itt.

```
#if DEBUG
 #if _KERNEL
 #define DEBUGP(format, args...) printk(KERN_INFO "xt_"_MODULE_NAME": "format”\n”, ##args)
 #else
 #define DEBUGP(format, args...) printf("# DEBUG: libxt_"_MODULE_NAME": "format"\n", ##args)
 #endif
#else
#define DEBUGP(format, args...)
#endif
```

Ha bekapcsoltuk a hibakeresést, akkor a fenti makró segítségével üzeneteket írhatunk a rendszernaplóba, konzolra.

```
#define XT_IFACE_FLAGCOUNT 11

enum
{
        XT_IFACE_UP             = 1 << 0,
        XT_IFACE_BROADCAST      = 1 << 1,

...

        XT_IFACE_DORMANT        = 1 << 10,
        XT_IFACE_IFACE          = 1 << 15,
};
```

A modul tizenegy állapotjelzőt (*XT_IFACE_FLAGCOUNT*) és azok inverzét tudja ellenőrizni egy adott interfészen. (A tizenkettedik *XT_IFACE_IFACE* opció a felhasználói tér programjában jelzi, hogy megadtunk-e már egy érvényes interfésznevet.) Mivel ezeknek az állapotoknak többféle kombinációját is figyelembe kell venni, ezért minden egyes állapot és inverze külön jelzőbitként kerül eltárolásra. A jelzőbitek megadásánál a biteken végzett balra eltolás funkciót használjuk.

A kernel részben egy cikluson keresztül fogjuk majd ellenőrizni az állapotbiteket, amihez egy keresőtáblát alkalmazunk. Ennek a struktúráját így adjuk meg:

```
struct xt_iface_flag_pairs
{
        uint16_t iface_flag;
        uint32_t iff_flag;
};
```

A kernel és a felhasználói tér közötti kommunikációt modulunkban egy viszonylag kis struktúrán keresztül (*xt_iface_match_info*) valósítjuk meg. Ebben eltároljuk, az interfész nevét (*ifname*), illetve hogy milyen állapotaira (*flags*), inverz állapotaira (*invflags*). vagyunk kíváncsiak:

```
struct xt_iface_match_info
{
        char ifname[IFNAMSIZ];
        uint16_t flags;
        uint16_t invflags;
};
```

6.2.2. Kernel rész

A közös fejléc fájl elkészítése után eljutottunk a modul legfontosabb részéhez, a kernel részhez. Ennek nagyon fontos a hibamentessége, hiszen a legkisebb probléma esetén is rendszerleállás következhet be. A fájlt nevezzük el *xt_iface.c*-nek.

```
#include <linux/if.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/netdevice.h>
#include <linux/skbuff.h>
#include <linux/netfilter/x_tables.h>
#include <linux/netfilter/xt_iface.h>
```

A modulhoz szükséges fejlécek listájában megtaláljuk a korábban elkészített saját fejlécünket (*xt_iface.h*) és egy nagyon fontos fejléc fájlt, ami a NETFILTER része. Ez az *x_tables.h* fájl, amit a */usr/src/linux/include/linux/netfilter/x_tables.h* útvonalon találunk meg.

Ez a fájl adja meg, hogy milyen struktúrákat kell használnunk ahhoz, hogy a modult a kernelbe tudjuk illeszteni. Egyezőségeknél a következő fejlécrészlet írja le a csatlakozási felületet: (Csak a fontos változókat meghagyva.)

```
struct xt_match {
        const char name[XT_FUNCTION_MAXNAMELEN-1];

        bool (*match)(const struct sk_buff *skb,
                      const struct net_device *in,
                      const struct net_device *out,
                      const struct xt_match *match,
                      const void *matchinfo,
                      int offset,
                      unsigned int protoff,
                      bool *hotdrop);

        bool (*checkentry)(const char *tablename,
                           const void *ip,
                           const struct xt_match *match,
                           void *matchinfo,
                           unsigned int hook_mask);

        void (*destroy)(const struct xt_match *match, void *matchinfo);

        unsigned int matchsize;
        unsigned short family;
        u_int8_t revision;
};
```

A struktúra összes változójának kitöltése ugyan nem kötelező, de az előző struktúrarészlet a minimálisan szükséges.
A *name* változóba azt a maximum 28 karakter hosszúságú nevet kell elhelyeznünk, amivel az IPTABLES-ben hivatkozni szeretnénk a modulra.
A modul gerincét a *match*, *checkentry* és a *destroy* nevű úgynevezett *callback* (továbbiakban: visszahívási) függvények adják. Ezeket a függvényeket a rendszer az egyezőség ellenőrzésekor, betöltésekor illetve felszabadításakor hívja meg.
A *matchsize* az előző fejlécfájlunkban definiált *xt_iface_match_info* struktúra méretét igényli. A *family* változó adja meg, hogy modulunkat melyik protokollcsaládhoz kapcsoljuk. *(ipv4/ipv6/arp/…)*
Végül a *revision* változó informálja a kernelt arról, hogy modulunk melyik verzióját szeretnénk használni. A kernel és a felhasználói program közötti kommunikáció csak akkor sikeres, ha mindkét fél ugyanazt a *name*, *revision*, *family*, *matchsize* négyest használja.
Mielőtt a struktúra változóinak feltöltését elvégeznénk, el kell készítenünk a visszahívási függvényeket, illetve a modul néhány tulajdonságát is közölnünk kell a rendszerrel.
Modulunkat lehetőség van nem csak *iptables*, hanem *ip6tables* alá is regisztrálni. Fontos információ még a modul fejlesztőjének neve, illetve a modul „álnevei”, valamint a licenc típusa.

Ezeket az információkat az alábbi makró-hívások segítenek beállítani:

```
MODULE_AUTHOR("Gáspár Lajos <gaspar.lajos@glsys.eu>");
MODULE_DESCRIPTION("Xtables: iface match module");
MODULE_LICENSE("GPL");
MODULE_ALIAS("ipt_iface");
MODULE_ALIAS("ip6t_iface");
```

Ezzel tudattuk a rendszerrel, hogy a modul *ipt_iface* és *ip6t_iface* néven is elérhető lesz. A továbbiakban még szükségünk lesz a korábbiakban említett keresőtáblára, amiben a mi jelzőbitjeinket megfeleltetjük a kernel-ben használatosakkal:

```
static struct xt_iface_flag_pairs xt_iface_lookup[XT_IFACE_FLAGCOUNT] =
{
        {.iface_flag = XT_IFACE_UP,             .iff_flag = IFF_UP},
        {.iface_flag = XT_IFACE_BROADCAST,      .iff_flag = IFF_BROADCAST},

...

        {.iface_flag = XT_IFACE_LOWER_UP,       .iff_flag = IFF_LOWER_UP},
        {.iface_flag = XT_IFACE_DORMANT,        .iff_flag = IFF_DORMANT},
};
```

Most már elkészíthetjük a legfontosabb visszahívási függvényt, a *match*-ot:

```
static bool xt_iface_match(const struct sk_buff *skb,
                           const struct net_device *in,
                           const struct net_device *out,
                           const struct xt_match *match,
                           const void *matchinfo,
                           int offset,
                           unsigned int protoff, bool * hotdrop)
{
        const struct xt_iface_match_info *info = matchinfo;
        struct net_device *dev;
        bool retval;
        int i;
        DEBUGP("match...");
        DEBUGP("Interface: %s", info->ifname);
        retval =
            ((dev = dev_get_by_name(&init_net, info->ifname)) != NULL);
        if (retval) {

...

                for (i=0; (i<XT_IFACE_FLAGCOUNT) && (retval); i++)
                        {
                        if (info->flags & xt_iface_lookup[i].iface_flag)
                                retval = retval && (dev->flags & xt_iface_lookup[i].iff_flag);
                        if (info->invflags & xt_iface_lookup[i].iface_flag)
                                retval = retval && !(dev->flags & xt_iface_lookup[i].iff_flag);
                        }
                dev_put(dev);
        }
        return retval;
}
```

Az előbbi forráskódot három érdekes részre lehet választani. Az első a visszahívási függvény paraméterlistája. Ebben a *matchinfo* változót találjuk, ami a felhasználói program segítségével beállított *xt_iface_match_info* struktúrát takarja, és ebben az interfész nevét, illetve azokat az opciókat, amikre kíváncsiak vagyunk. Tehát itt kapjuk meg a felhasználói programtól, hogy milyen opciók állapotát kell lekérdeznünk a kernelben. Ehhez az *info* nevű struktúra bevezetésével jutunk hozzá.
A második érdekes rész, amikor *dev_get_by_name* funkcióval lekérdezzük az interfész állapotát. Ha nem találunk ilyen eszközt, akkor a rendszer *NULL* értéket ad vissza a függvényhívás után. Ebben az esetben a *retval* változó értéke hamis lesz, és ezzel az értékkel vissza is tér a *match* funkciónk a kernelhez. Azaz az a szabály, amelyben alkalmaztuk az egyezőségünket, nem fog érvényesülni.
Az utolsó érdekesség a fenti forráskódban akkor hajtódik végre, ha a *dev_get_by_name* megtalálja az általunk keresett interfészt. Ekkor egy *for* ciklus segítségével következik az *info* struktúra *flags* illetve *invflags* váltózói alapján az eszköz állapotbitjeinek vizsgálata. Itt hívjuk segítségül a korábbi keresőtáblát.
Fontos, ha a *dev_get_by_name* megtalálta az eszközt, akkor a vizsgálatok végén a *dev_put* függvénnyel zárjuk le a hozzáférést az interfész adataihoz.

A következő visszahívási funkció, amit el kell készítenünk a *checkentry*. Feladata, hogy egy végső ellenőrzést tegyünk az egyezőség rendszerbe illesztése előtt, illetve ha szükséges, még más modulokat is be tudjunk tölteni ebben a funkcióban.

```
static bool xt_iface_checkentry(const char *tablename,
                                const void *ip,
                                const struct xt_match *match,
                                void *matchinfo, unsigned int hook_mask)
{
        DEBUGP("checkentry...");
        return true;
}
```

Mivel ilyen feladatok elvégzésére nincs szükségünk, ezért ez a funkció igaz értékkel tér vissza a kernelhez, és engedélyezi a modul kernelbe illesztését.

Az utolsó kernelfunkciónk a *destroy*, amiben a *checkentry* ellentettjeként a modul eltávolítása során erőforrás felszabadítást végezhetünk.

```
static void xt_iface_destroy(const struct xt_match *match, void *matchinfo)
{
        DEBUGP("destroy...");
}
```

Mivel plusz erőforrásokat nem használtunk, ezért ez a funkció elkészítése nem kötelező. Elhagyásával igazából nem nyerünk semmit, viszont egy későbbi fejlesztésnél, kiegészítésnél elfeledkezhetünk róla, ami rejtett hibákat okozhat.

Most már ki tudjuk tölteni az *xt_match* struktúra változóit mindkét protokollcsaládhoz *(ipv4, ipv6)*:

```
static struct xt_match xt_iface_match_reg[] __read_mostly = {
        {
         .name          = _MODULE_NAME,
         .match         = xt_iface_match,
         .checkentry    = xt_iface_checkentry,
         .destroy       = xt_iface_destroy,
         .me            = THIS_MODULE,
         .data          = 0,
         .matchsize     = XT_ALIGN(sizeof(struct xt_iface_match_info)),
         .family        = AF_INET,
         .revision      = _MODULE_REVISION,
         },
        {
         .name          = _MODULE_NAME,
         .match         = xt_iface_match,
         .checkentry    = xt_iface_checkentry,
         .destroy       = xt_iface_destroy,
         .me            = THIS_MODULE,
         .data          = 0,
         .matchsize     = XT_ALIGN(sizeof(struct xt_iface_match_info)),
         .family        = AF_INET6,
         .revision      = _MODULE_REVISION,
         }
};
```

Ahhoz, hogy a modul betöltése után használni tudjuk az egyezőséget, még regisztrálnunk kell a NETFILTER-ben. Ehhez az *xt_register_matches* funkciót használjuk:

```
static int __init xt_iface_match_init(void)
{
        DEBUGP("init...\n");
        return xt_register_matches(xt_iface_match_reg,
                                   ARRAY_SIZE(xt_iface_match_reg));
}
```

Modulunk eltávolításakor az ellenkező funkciót kell végrehajtani:

```
static void __exit xt_iface_match_exit(void)
{
        DEBUGP("exit...\n");
        xt_unregister_matches(xt_iface_match_reg,
                              ARRAY_SIZE(xt_iface_match_reg));
}
```

Ahhoz, hogy ezek a funkciók a modul betöltésekor automatikusan végrehajtódjanak, két makrót még be kell illesztenünk a forráskódba:

```
module_init(xt_iface_match_init);
module_exit(xt_iface_match_exit);
```

Ezzel elkészítettük a kernelben futó részét a kiegészítésnek. Azonban ahhoz, hogy működésbe tudjuk hozni a modult, még a felhasználói felületről utasítanunk kell az IPTABLES-t. Ehhez viszont a következő részre lesz szükségünk.

6.2.3. Felhasználói rész

Ugyanúgy, ahogy a kernelmodulnál, itt is a szükséges fejlécfájlok beillesztésével kezdjük programunkat. Ez a program a *libxt_iface.c* nevet fogja kapni.

```
#include <getopt.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <xtables.h>
#include <linux/netfilter/xt_iface.h>
```

A felhasználói tér programjainál az */usr/src/iptables/include/xtables.h* fejlécfájl tartalmazza a rendszerbeillesztéshez szükséges információkat: (Itt is csak a fontosabb változókat meghagyva.)

```
struct xtables_match{

        xt_chainlabel name;

        u_int8_t revision;

        u_int16_t family;

        const char *version;

        size_t size;

        size_t userspacesize;

        void (*help)(void);

        void (*init)(struct xt_entry_match *m);

        int (*parse)(int c, char **argv, int invert, unsigned int *flags,
                     const void *entry,
                     struct xt_entry_match **match);

        void (*final_check)(unsigned int flags);

        void (*print)(const void *ip,
                      const struct xt_entry_match *match, int numeric);

        void (*save)(const void *ip, const struct xt_entry_match *match);

        const struct option *extra_opts;
};
```

Az első négy változó megfelel a kernelmodulban használt változóknak. Ezek egyezősége esetén lehet használni a modult. A *size* és *userspacesize* változók a modulunk által felhasznált tárterület méretét adják meg. Ezek után következnek a funkciók definíciói.

A funkciók tárgyalása előtt nézzük át, milyen opciókkal lehet majd meghívni a modulunkat. Ehhez az *xt_iface_match_opts* struktúrát kell feltöltenünk, amit majd az *extra_opts*-ban kell átadnunk az IPTABLES-nek:

```
static struct option xt_iface_match_opts[] = {
        {.name = "iface",       .has_arg = true,        .flag = 0,      .val = 'i'},
        {.name = "up",          .has_arg = false,       .flag = 0,      .val = 'u'},
        {.name = "down",        .has_arg = false,       .flag = 0,      .val = 'U'},    // not up
        {.name = "broadcast",   .has_arg = false,       .flag = 0,      .val = 'b'},
        {.name = "loopback",    .has_arg = false,       .flag = 0,      .val = 'l'},
        {.name = "pointopoint", .has_arg = false,       .flag = 0,      .val = 'p'},
        {.name = "pointtopoint",.has_arg = false,       .flag = 0,      .val = 'p'},    // eq
pointopoint
        {.name = "running",     .has_arg = false,       .flag = 0,      .val = 'r'},
        {.name = "noarp",       .has_arg = false,       .flag = 0,      .val = 'n'},
        {.name = "arp",         .has_arg = false,       .flag = 0,      .val = 'N'},    // not noarp
        {.name = "promisc",     .has_arg = false,       .flag = 0,      .val = 'o'},
        {.name = "promiscous",  .has_arg = false,       .flag = 0,      .val = 'o'},    // eq
promisc
        {.name = "multicast",   .has_arg = false,       .flag = 0,      .val = 'm'},
        {.name = "dynamic",     .has_arg = false,       .flag = 0,      .val = 'd'},
        {.name = "lower_up",    .has_arg = false,       .flag = 0,      .val = 'w'},
        {.name = "dormant",     .has_arg = false,       .flag = 0,      .val = 'a'},
        {.name = NULL},
};
```

Ebben a tömbben találunk néhány egymással ellentétes opciót is. Ezek a felhasználó munkájának megkönnyítésére lettek beépítve. (Például a *--down* opció megfelel a *! --up* funkciónak.) Azért hogy ezekről az opciókról rövid leírást adhassunk a modul használatát segítendő, el kell készítenünk a *help* funkciót:

```
static void xt_iface_match_help(void)
{
        printf(_MODULE_NAME " match v%s rev:%#2x options:\n"
               " --iface interface\t\tName of interface\n"
               " [!] --up\n"
               " [!] --down\t\t\tmatch if UP flag (not) set\n"
               " [!] --broadcast\t\tmatch if BROADCAST flag (not) set\n"
               " [!] --loopback\t\t\tmatch if LOOPBACK flag (not) set\n"
               " [!] --pointopoint\n"
               " [!] --pointtopoint\t\tmatch if POINTOPOINT flag (not) set\n"
               " [!] --running\t\t\tmatch if RUNNING flag (not) set\n"
               " [!] --noarp\n"
               " [!] --arp\t\t\tmatch if NOARP flag (not) set\n"
               " [!] --promisc\n"
               " [!] --promiscous\t\tmatch if PROMISC flag (not) set\n"
               " [!] --multicast\t\tmatch if MULTICAST flag (not) set\n"
               " [!] --dynamic\t\t\tmatch if DYNAMIC flag (not) set\n"
               " [!] --lower_up\t\t\tmatch if LOWER_UP flag (not) set\n"
               " [!] --dormant\t\t\tmatch if DORMANT flag (not) set\n",
               IPTABLES_VERSION, _MODULE_REVISION);
}
```

A modul tartalmaz három csak belső használatra szánt funkciót is. Az első funkció segít a szabályok mentésekor, illetve a konzolra listázáskor:

```
static void xt_iface_print_option(const struct xt_iface_match_info *info,
                        const unsigned int option,
                        const char *command)
{
        DEBUGP("print_option... %s", command);
        if (info->flags & option) printf(" --%s", command);
        if (info->invflags & option) printf(" ! --%s", command);
}
```

A másodikkal a szabályban szereplő jelzőbitek eltárolását segítjük: (Ha korábban már szerepelt a szabályban az adott jelzőbit, akkor hibaüzenetet generálva megszakíttatjuk a program futását.)

```
static void xt_iface_setflag(struct xt_iface_match_info *info,
                             unsigned int *flags,
                             int invert,
                             uint16_t flag,
                             const char *command)
{
        DEBUGP("setflag... %s", command);
        if (*flags & flag)
                exit_error(PARAMETER_PROBLEM,
                           "xt_iface: \"--%s\" flag already specified!",
                           command);
        if (invert)
                info->invflags |= flag;
                else
                info->flags |= flag;
        *flags |= flag;
}
```

Végül a harmadik feladata az interfész nevének ellenőrzése:

```
static bool xt_iface_valid_interface_name(const char *name)
{
        char invalid_chars[] = ".+!*";
        DEBUGP("valid_interface_name... %d %d", strcspn(name, invalid_chars), strlen(name));
        return !((strlen(name) >= IFNAMSIZ) || (strcspn(name, invalid_chars) != strlen(name)));
}
```

A hossz és a meg nem engedett karakterek leellenőrzésével biztosítjuk, hogy ne legyenek félregépelések a szabályok bevitelekor.

A felhasználói tér moduljának legfontosabb része a *parse* funkció. Ebben tudjuk leellenőrizni, illetve előkészíteni a kernel modulnak szóló információkat arról, hogy az adott interfész mely állapotaira vagyunk kíváncsiak. Itt kell kezelnünk a korábbiakban említett ellentétes opciókat is:

```
static int xt_iface_match_parse(int c,
                                 char **argv,
                                 int invert,
                                 unsigned int *flags,
                                 const void *entry,
                                 struct xt_entry_match **match)
{
        DEBUGP("parse... c:%c invert:%x", c, invert);
        struct xt_iface_match_info *info = (void *) (*match)->data;
        switch (c) {
        case 'U':
                c = 'u';
                invert = !invert;
                break;
        case 'N':
                c = 'n';
                invert = !invert;
                break;
        }
        switch (c) {
        case 'i':               // iface
                if (*flags & XT_IFACE_IFACE)
                        exit_error(PARAMETER_PROBLEM,
                                   "xt_iface: Interface name already specified!");
                if (!xt_iface_valid_interface_name(optarg))
                        exit_error(PARAMETER_PROBLEM,
                                   "xt_iface: Invalid interface name!");
                strcpy(info->ifname, optarg);
                *flags |= XT_IFACE_IFACE;
                return 1;
        case 'u':               //up
                xt_iface_setflag(info, flags, invert, XT_IFACE_UP, "up");
                return 1;
        case 'b':               // broadcast
                xt_iface_setflag(info, flags, invert, XT_IFACE_BROADCAST, "broadcast");
                return 1;

...

        case 'a':               // dormant
                xt_iface_setflag(info, flags, invert, XT_IFACE_DORMANT, "dormant");
                return 1;
        default:
                return 0;
        }
}
```

Következő funkciónk a *check*. Feladata, hogy mielőtt a szabályt a rendszer befogadná, még egy végső ellenőrzést biztosítson. (Előfordulhat ugyanis, hogy paraméterek nélkül próbáljuk meghívni a modult, ami hibát eredményez.)

```
static void xt_iface_match_final_check(unsigned int flags)
{
        DEBUGP("final_check...");
        if (!(flags & XT_IFACE_IFACE))
                exit_error(PARAMETER_PROBLEM,
                           "iface: You must specify an interface !");
        if ((flags == 0) || (flags == XT_IFACE_IFACE))
                exit_error(PARAMETER_PROBLEM,
                           "iface: You must specify at least one option !");
}
```

A felhasználó kérésére a rendszer állapotjelentéseket tud generálni. Ezért a modulunkat is fel kell készíteni a beállított opciók konzolra történő írására:

```
static void xt_iface_match_print(const void *ip,
                                 const struct xt_entry_match *match,
                                 int numeric)
{
        DEBUGP("print...");
        const struct xt_iface_match_info *info =
            (const void *) match->data;
        printf("--iface %s", info->ifname);
        xt_iface_print_option(info, XT_IFACE_UP, "up");
        xt_iface_print_option(info, XT_IFACE_BROADCAST, "broadcast");
        xt_iface_print_option(info, XT_IFACE_LOOPBACK, "loopback");
        xt_iface_print_option(info, XT_IFACE_POINTOPOINT, "pointopoint");
        xt_iface_print_option(info, XT_IFACE_RUNNING, "running");
        xt_iface_print_option(info, XT_IFACE_NOARP, "noarp");
        xt_iface_print_option(info, XT_IFACE_PROMISC, "promisc");
        xt_iface_print_option(info, XT_IFACE_MULTICAST, "multicast");
        xt_iface_print_option(info, XT_IFACE_DYNAMIC, "dynamic");
        xt_iface_print_option(info, XT_IFACE_LOWER_UP, "lower_up");
        xt_iface_print_option(info, XT_IFACE_DORMANT, "dormant");
        printf(" ");
}
```

A *save* funkció gyakorlatilag megegyezik a print funkcióval, mivel ez az *iptables-save* és *iptables-restore* parancsoknak megfelelő kimenetet produkál:

```
static void xt_iface_match_save(const void *ip,
                                const struct xt_entry_match *match)
{
        DEBUGP("save...");
        xt_iface_match_print(ip, match, 0);
}
```

Ha a modul, amit fejlesztünk, igényel különféle erőforrásokat a működéséhez, akkor azokat inicializálni kell. Erre a feladatra készíthetjük fel az *init* funkciót. (A bemutatott modulban nem használunk ilyen funkciókat.)

```
static void xt_iface_match_init(struct xt_entry_match *m)
{
        DEBUGP("init...");
}
```

Miután eddig eljutottunk, meg kell mondanunk az IPTABLES felületnek, hogy milyen funkciókat biztosít a modul. Mivel *ipv4* és *ipv6* alatt is használható lesz a kiegészítőnk, ezért ezekhez a rendszerekhez is regisztrálnunk kell:

```
static struct xtables_match xt_iface_match_reg = {
        .name           = _MODULE_NAME,
        .revision       = _MODULE_REVISION,
        .version        = IPTABLES_VERSION,
        .family         = AF_INET,
        .size           = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .userspacesize  = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .help           = xt_iface_match_help,
        .init           = xt_iface_match_init,
        .parse          = xt_iface_match_parse,
        .final_check    = xt_iface_match_final_check,
        .print          = xt_iface_match_print,
        .save           = xt_iface_match_save,
        .extra_opts     = xt_iface_match_opts,
};

static struct xtables_match xt_iface_match6_reg = {
        .name           = _MODULE_NAME,
        .revision       = _MODULE_REVISION,
        .version        = IPTABLES_VERSION,
        .family         = AF_INET6,
        .size           = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .userspacesize  = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .help           = xt_iface_match_help,
        .init           = xt_iface_match_init,
        .parse          = xt_iface_match_parse,
        .final_check    = xt_iface_match_final_check,
        .print          = xt_iface_match_print,
        .save           = xt_iface_match_save,
        .extra_opts     = xt_iface_match_opts,
};
```

Az utolsó lépés a modul betöltésekor az IPTABLES-ben való regisztráció, amit külön kell minden protokollhoz elvégeznünk.

```
void _init(void)
{
        DEBUGP("_init...");
        xtables_register_match(&xt_iface_match_reg);
        xtables_register_match(&xt_iface_match6_reg);
}
```

6.3. Fordítás, telepítés, tesztelés

Az elkészült forráskódokat helyezzük el a kernelben, illetve az IPTABLES alkönyvtáraiban:

```
cp libxt_iface.c /us/src/iptables/extensions/
cp xt_iface.c /usr/src/linux/net/netfilter/
cp xt_iface.h /usr/src/linux/include/linux/netfilter/
```

Adjunk még hozzá a kernel forrásában lévő *Makefie*-hoz egy sort:

```
echo 'obj-m += xt_iface.o' >>/usr/src/linux/net/netfilter/Makefile
```

Ezek után már csak le kell fordítani az új kernelmodult és az IPTABLES forrást:

```
cd /usr/src/linux
make modules
cd /usr/src/iptables
make
```

Ha minden gond nélkül sikerült lefordítanunk a modult, akkor már csak egy feladat vár ránk, a telepítés. Az alábbi pár sor elvégzi ezeket a lépéseket.

```
cd /usr/src/linux
make modules_install
cd /usr/src/iptables
make install
```

A kernel automatikusan betölti a szükséges modulokat, ha megpróbáljuk használni őket. Ezért ki is tudjuk próbálni az elkészült kiegészítőt:

```
iptables -t raw -A OUTPUT -j ACCEPT -o lo -m iface --iface lo --up
```

A működést az *iptables -t raw -vnL OUTPUT* paranccsal ellenőrizhetjük:

```
Chain OUTPUT (policy ACCEPT 8204K packets, 5112M bytes)
 pkts bytes target     prot opt in     out     source           destination
   87  8481 ACCEPT     all  --  *      lo      0.0.0.0/0        0.0.0.0/0           --iface lo --up
```

7. ÖSSZEGZÉS

A NETFILTER/IPTABLES páros a lehetőségek szinte végtelen tárházát adja arra, hogy különféle eseményeket, információkat vizsgáljunk a tűzfalak szabályrendszereinek segítségével. Kihívás volt számomra olyan kiegészítőt kitalálni, ami egyértelműen hasznosan kiegészítette volna a rendszert. Mégis úgy gondolom, adódhat olyan helyzet, ahol ez az egyezőség meggyorsíthatja, leegyszerűsítheti a tűzfal működését.

Az általam fejlesztett modult olyan esetekben lehet felhasználni, amelyeknél az interfész állapota hatással van a rendszer további működésére. Beágyazott rendszerek esetében ez az információ akár az erőforrások kihasználásának mikéntjét is befolyásolhatja.

Ilyenre lehet példa akár egy úgynevezett „*failover*” konfiguráció, ahol az eszközök meg tudják állapítani a másik fél kiesését az által, hogy *cross-link* kábellel összekötött interfészeik működőképességét ellenőrzik.

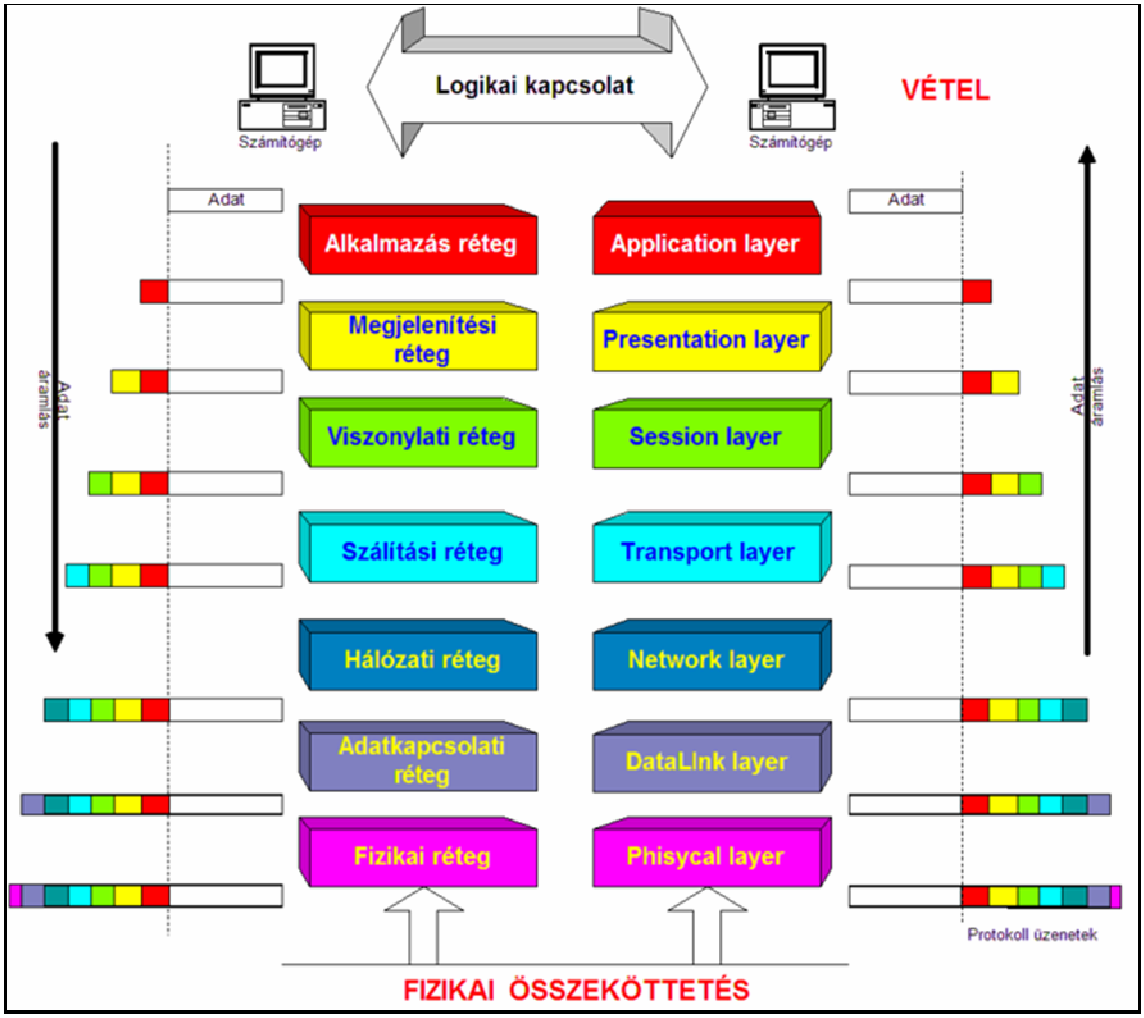
Érdekes megoldás lehet az a terheléselosztó rendszer is, ahol a kapcsolatok működőképessége függvényében irányítjuk az új kapcsolatokat. Egy ilyen komplex feladat az az eset, ahol három kapcsolattal csatlakozunk az Internetre, és ezek közül az egyik dedikáltan a levelezés kiszolgálására van fenntartva. Ha ez a vonal valami oknál fogva nem működik, akkor ettől függetlenül meg kell oldani a levelek kifelé küldésének problémáját. Ebben az esetben a Linux sávszélesség korlátozó mechanizmusának segítségével és a *statistic* modul felhasználásával szétoszthatjuk a fennmaradó két kapcsolat között a terhelést.

Természetesen a modul jövőbeli fejlesztésébe olyan funkciók is bekerülhetnek, amelyek majd szélesebb felhasználási területet biztosítanak. Mivel a modul elkészítésekor a funkcionalitás volt számomra a fő szempont, ezért biztosnak tartom, hogy egyes funkciókat lehetne jobban, „szebben” is megoldani. Ugyanakkor törekedtem arra, hogy a hibák lehetőségét a minimálisra csökkentsem.

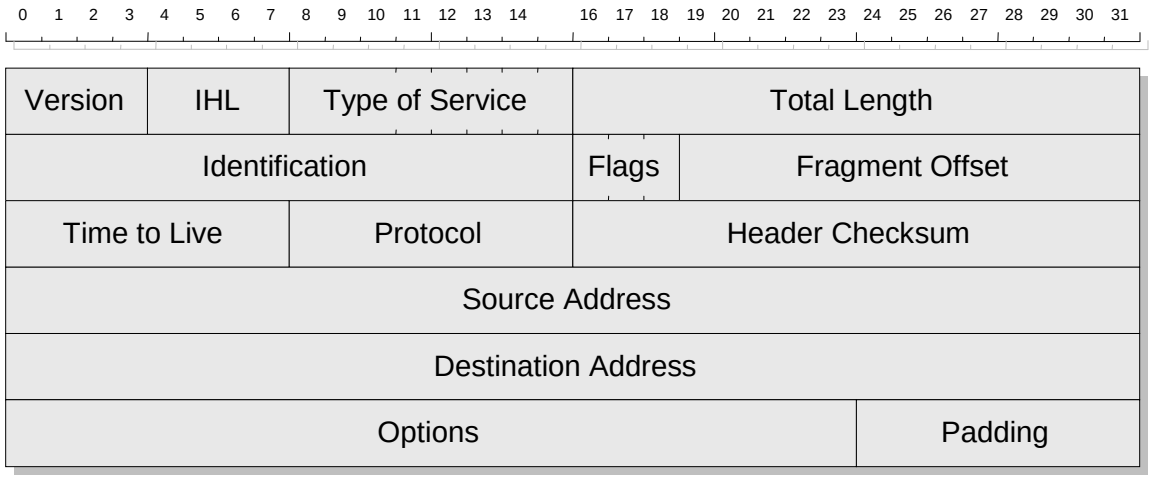
Remélem, hogy sikerült egy hasznos kiegészítőt alkotnom a NETFILTER/IPTABLES rendszerhez.

8. MELLÉKLET

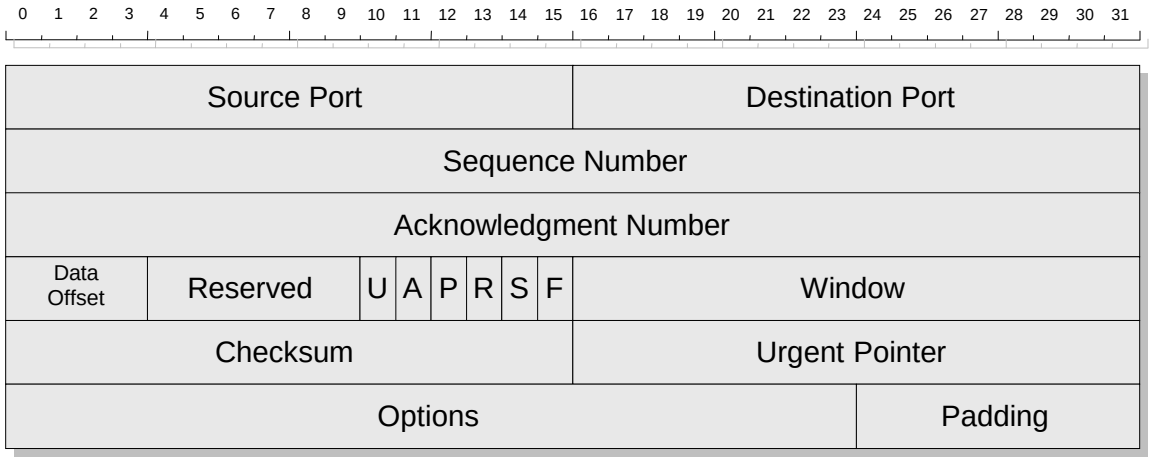
8.1. Ábrák



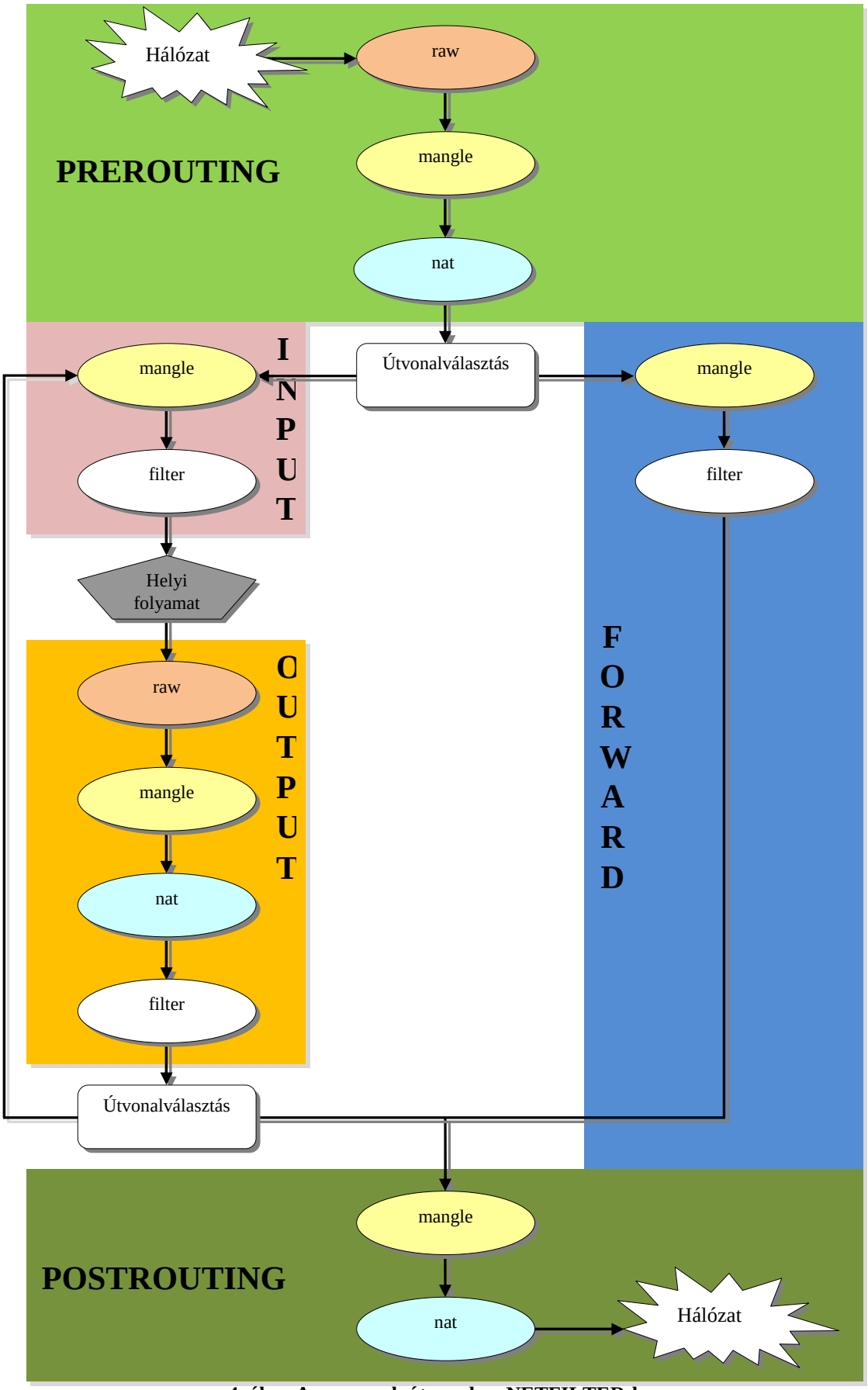
1. ábra ISO-OSI modell



2. ábra IP fejléc



3. ábra TCP fejléc



4. ábra A csomagok útvonala a NETFILTER-ben

8.2. Forráskód

8.2.1. Közös fejléc fájl

```
#ifndef _LINUX_NETFILTER_XT_IFACE_H
#define _LINUX_NETFILTER_XT_IFACE_H 1

#define DEBUG 0
#define _MODULE_NAME "iface"
#define _MODULE_REVISION 0

#if DEBUG
#if _KERNEL
#define DEBUGP(format, args...) printk(KERN_INFO "xt_"_MODULE_NAME": "format"\n", ##args)
#else
#define DEBUGP(format, args...) printf("# DEBUG: libxt_"_MODULE_NAME": "format"\n", ##args)
#endif
#else
#define DEBUGP(format, args...)
#endif

#define XT_IFACE_FLAGCOUNT 11

enum
{
        XT_IFACE_UP             = 1 << 0,
        XT_IFACE_BROADCAST      = 1 << 1,
        XT_IFACE_LOOPBACK       = 1 << 2,
        XT_IFACE_POINTOPOINT    = 1 << 3,
        XT_IFACE_RUNNING        = 1 << 4,
        XT_IFACE_NOARP          = 1 << 5,
        XT_IFACE_PROMISC        = 1 << 6,
        XT_IFACE_MULTICAST      = 1 << 7,
        XT_IFACE_DYNAMIC        = 1 << 8,
        XT_IFACE_LOWER_UP       = 1 << 9,
        XT_IFACE_DORMANT        = 1 << 10,
        XT_IFACE_IFACE          = 1 << 15,
};

struct xt_iface_flag_pairs
{
        uint16_t iface_flag;
        uint32_t iff_flag;
};

struct xt_iface_match_info
{
        char ifname[IFNAMSIZ];
        uint16_t flags;
        uint16_t invflags;
};

#endif
```

8.2.2. Kernel rész

```
/*
 *      xt_iface - kernel module to match interface state flags
 *
 *      Original author: Gáspár Lajos <gaspar.lajos@glsys.eu>
 */

#define _KERNEL 1

#include <linux/if.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/netdevice.h>
#include <linux/skbuff.h>
#include <linux/netfilter/x_tables.h>
#include <linux/netfilter/xt_iface.h>

MODULE_AUTHOR("Gáspár Lajos <gaspar.lajos@glsys.eu>");
MODULE_DESCRIPTION("Xtables: iface match module");
MODULE_LICENSE("GPL");
MODULE_ALIAS("ipt_iface");
MODULE_ALIAS("ip6t_iface");
//MODULE_ALIAS("arpt_iface");

static struct xt_iface_flag_pairs xt_iface_lookup[XT_IFACE_FLAGCOUNT] =
{
        {.iface_flag = XT_IFACE_UP,             .iff_flag = IFF_UP},
        {.iface_flag = XT_IFACE_BROADCAST,      .iff_flag = IFF_BROADCAST},
        {.iface_flag = XT_IFACE_LOOPBACK,       .iff_flag = IFF_LOOPBACK},
        {.iface_flag = XT_IFACE_POINTOPOINT,    .iff_flag = IFF_POINTOPOINT},
        {.iface_flag = XT_IFACE_RUNNING,        .iff_flag = IFF_RUNNING},
        {.iface_flag = XT_IFACE_NOARP,          .iff_flag = IFF_NOARP},
        {.iface_flag = XT_IFACE_PROMISC,        .iff_flag = IFF_PROMISC},
        {.iface_flag = XT_IFACE_MULTICAST,      .iff_flag = IFF_MULTICAST},
        {.iface_flag = XT_IFACE_DYNAMIC,        .iff_flag = IFF_DYNAMIC},
        {.iface_flag = XT_IFACE_LOWER_UP,       .iff_flag = IFF_LOWER_UP},
        {.iface_flag = XT_IFACE_DORMANT,        .iff_flag = IFF_DORMANT},
};

static bool xt_iface_match(const struct sk_buff *skb,
                           const struct net_device *in,
                           const struct net_device *out,
                           const struct xt_match *match,
                           const void *matchinfo,
                           int offset,
                           unsigned int protoff, bool * hotdrop)
{
        const struct xt_iface_match_info *info = matchinfo;
        struct net_device *dev;
        bool retval;
        int i;
        DEBUGP("match...");
        DEBUGP("Interface: %s", info->ifname);
        retval =
            ((dev = dev_get_by_name(&init_net, info->ifname)) != NULL);
        if (retval) {
#if DEBUG
                DEBUGP("dev->flags: %#8x", dev->flags);
                if (dev->flags & IFF_UP)
                        DEBUGP("            %#8x (UP)", IFF_UP);
                if (dev->flags & IFF_BROADCAST)
                        DEBUGP("            %#8x (BROADCAST)", IFF_BROADCAST);
                if (dev->flags & IFF_LOOPBACK)
                        DEBUGP("            %#8x (LOOPBACK)", IFF_LOOPBACK);
                if (dev->flags & IFF_POINTOPOINT)
                        DEBUGP("            %#8x (POINTOPOINT)", IFF_POINTOPOINT);
                if (dev->flags & IFF_RUNNING)
                        DEBUGP("            %#8x (RUNNING)", IFF_RUNNING);
                if (dev->flags & IFF_NOARP)
```

```
                        DEBUGP("            %#8x (NOARP)", IFF_NOARP);
                if (dev->flags & IFF_PROMISC)
                        DEBUGP("            %#8x (PROMISC)", IFF_PROMISC);
                if (dev->flags & IFF_MULTICAST)
                        DEBUGP("            %#8x (MULTICAST)", IFF_MULTICAST);
                if (dev->flags & IFF_DYNAMIC)
                        DEBUGP("            %#8x (DYNAMIC)", IFF_DYNAMIC);
                if (dev->flags & IFF_LOWER_UP)
                        DEBUGP("            %#8x (LOWER_UP)", IFF_LOWER_UP);
                if (dev->flags & IFF_DORMANT)
                        DEBUGP("            %#8x (DORMANT)", IFF_DORMANT);
#endif
                for (i=0; (i<XT_IFACE_FLAGCOUNT) && (retval); i++)
                        {
                        if (info->flags & xt_iface_lookup[i].iface_flag)
                                retval = retval && (dev->flags & xt_iface_lookup[i].iff_flag);
                        if (info->invflags & xt_iface_lookup[i].iface_flag)
                                retval = retval && !(dev->flags & xt_iface_lookup[i].iff_flag);
                        }
                dev_put(dev);
        }
        return retval;
}

static bool xt_iface_checkentry(const char *tablename,
                                const void *ip,
                                const struct xt_match *match,
                                void *matchinfo, unsigned int hook_mask)
{
        DEBUGP("checkentry...");
        return true;
}

static void xt_iface_destroy(const struct xt_match *match, void *matchinfo)
{
        DEBUGP("destroy...");
}

static struct xt_match xt_iface_match_reg[] __read_mostly = {
        {
         .name          = _MODULE_NAME,
         .match         = xt_iface_match,
         .checkentry    = xt_iface_checkentry,
         .destroy       = xt_iface_destroy,
         .me            = THIS_MODULE,
         .data          = 0,
         .matchsize     = XT_ALIGN(sizeof(struct xt_iface_match_info)),
         .family        = AF_INET,
         .revision      = _MODULE_REVISION,
         },
        {
         .name          = _MODULE_NAME,
         .match         = xt_iface_match,
         .checkentry    = xt_iface_checkentry,
         .destroy       = xt_iface_destroy,
         .me            = THIS_MODULE,
         .data          = 0,
         .matchsize     = XT_ALIGN(sizeof(struct xt_iface_match_info)),
         .family        = AF_INET6,
         .revision      = _MODULE_REVISION,
         }
};

static int __init xt_iface_match_init(void)
{
        DEBUGP("init...\n");
        return xt_register_matches(xt_iface_match_reg,
                                   ARRAY_SIZE(xt_iface_match_reg));
}

static void __exit xt_iface_match_exit(void)
```

```
{
        DEBUGP("exit...\n");
        xt_unregister_matches(xt_iface_match_reg,
                              ARRAY_SIZE(xt_iface_match_reg));
}

module_init(xt_iface_match_init);
module_exit(xt_iface_match_exit);
```

8.2.3. Felhasználói rész

```
/*
 * Shared library add-on to iptables to add interface state matching
 * support.
 *
 * (C) 2008 Gáspár Lajos <gaspar.lajos@glsys.eu>
 *
 * This program is released under the terms of GNU GPL version 2.
 */

#include <getopt.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <xtables.h>
#include <linux/netfilter/xt_iface.h>

static struct option xt_iface_match_opts[] = {
        {.name = "iface",       .has_arg = true,        .flag = 0,      .val = 'i'},
        {.name = "up",          .has_arg = false,       .flag = 0,      .val = 'u'},
        {.name = "down",        .has_arg = false,       .flag = 0,      .val = 'U'},    // not up
        {.name = "broadcast",   .has_arg = false,       .flag = 0,      .val = 'b'},
        {.name = "loopback",    .has_arg = false,       .flag = 0,      .val = 'l'},
        {.name = "pointopoint", .has_arg = false,       .flag = 0,      .val = 'p'},
        {.name = "pointtopoint",.has_arg = false,       .flag = 0,      .val = 'p'},    // eq
pointopoint
        {.name = "running",     .has_arg = false,       .flag = 0,      .val = 'r'},
        {.name = "noarp",       .has_arg = false,       .flag = 0,      .val = 'n'},
        {.name = "arp",         .has_arg = false,       .flag = 0,      .val = 'N'},    // not noarp
        {.name = "promisc",     .has_arg = false,       .flag = 0,      .val = 'o'},
        {.name = "promiscous",  .has_arg = false,       .flag = 0,      .val = 'o'},    // eq
promisc
        {.name = "multicast",   .has_arg = false,       .flag = 0,      .val = 'm'},
        {.name = "dynamic",     .has_arg = false,       .flag = 0,      .val = 'd'},
        {.name = "lower_up",    .has_arg = false,       .flag = 0,      .val = 'w'},
        {.name = "dormant",     .has_arg = false,       .flag = 0,      .val = 'a'},
        {.name = NULL},
};

static void xt_iface_print_option(const struct xt_iface_match_info *info,
                        const unsigned int option,
                        const char *command)
{
        DEBUGP("print_option... %s", command);
        if (info->flags & option) printf(" --%s", command);
        if (info->invflags & option) printf(" ! --%s", command);
}

static void xt_iface_setflag(struct xt_iface_match_info *info,
                             unsigned int *flags,
                             int invert,
                             uint16_t flag,
                             const char *command)
{
        DEBUGP("setflag... %s", command);
```

```
        if (*flags & flag)
                exit_error(PARAMETER_PROBLEM,
                           "xt_iface: \"--%s\" flag already specified!",
                           command);
        if (invert)
                info->invflags |= flag;
                else
                info->flags |= flag;
        *flags |= flag;
}

static bool xt_iface_valid_interface_name(const char *name)
{
        char invalid_chars[] = ".+!*";
        DEBUGP("valid_interface_name... %d %d", strcspn(name, invalid_chars), strlen(name));
        return !((strlen(name) >= IFNAMSIZ) || (strcspn(name, invalid_chars) != strlen(name)));
}

static void xt_iface_match_help(void)
{
        printf(_MODULE_NAME " match v%s rev:%#2x options:\n"
               " --iface interface\t\tName of interface\n"
               " [!] --up\n"
               " [!] --down\t\t\tmatch if UP flag (not) set\n"
               " [!] --broadcast\t\tmatch if BROADCAST flag (not) set\n"
               " [!] --loopback\t\t\tmatch if LOOPBACK flag (not) set\n"
               " [!] --pointopoint\n"
               " [!] --pointtopoint\t\tmatch if POINTOPOINT flag (not) set\n"
               " [!] --running\t\t\tmatch if RUNNING flag (not) set\n"
               " [!] --noarp\n"
               " [!] --arp\t\t\tmatch if NOARP flag (not) set\n"
               " [!] --promisc\n"
               " [!] --promiscous\t\tmatch if PROMISC flag (not) set\n"
               " [!] --multicast\t\tmatch if MULTICAST flag (not) set\n"
               " [!] --dynamic\t\t\tmatch if DYNAMIC flag (not) set\n"
               " [!] --lower_up\t\t\tmatch if LOWER_UP flag (not) set\n"
               " [!] --dormant\t\t\tmatch if DORMANT flag (not) set\n",
               IPTABLES_VERSION, _MODULE_REVISION);
}

static void xt_iface_match_init(struct xt_entry_match *m)
{
        DEBUGP("init...");
}

static int xt_iface_match_parse(int c,
                                char **argv,
                                int invert,
                                unsigned int *flags,
                                const void *entry,
                                struct xt_entry_match **match)
{
        DEBUGP("parse... c:%c invert:%x", c, invert);
        struct xt_iface_match_info *info = (void *) (*match)->data;
        switch (c) {
        case 'U':
                c = 'u';
                invert = !invert;
                break;
        case 'N':
                c = 'n';
                invert = !invert;
                break;
        }
        switch (c) {
        case 'i':               // iface
                if (*flags & XT_IFACE_IFACE)
                        exit_error(PARAMETER_PROBLEM,
                                   "xt_iface: Interface name already specified!");
                if (!xt_iface_valid_interface_name(optarg))
                        exit_error(PARAMETER_PROBLEM,
```

```
                                "xt_iface: Invalid interface name!");
                strcpy(info->ifname, optarg);
                *flags |= XT_IFACE_IFACE;
                return 1;
        case 'u':               //up
                xt_iface_setflag(info, flags, invert, XT_IFACE_UP, "up");
                return 1;
        case 'b':               // broadcast
                xt_iface_setflag(info, flags, invert, XT_IFACE_BROADCAST, "broadcast");
                return 1;
        case 'l':               // loopback
                xt_iface_setflag(info, flags, invert, XT_IFACE_LOOPBACK, "loopback");
                return 1;
        case 'p':               // pointopoint
                xt_iface_setflag(info, flags, invert, XT_IFACE_POINTOPOINT, "pointopoint");
                return 1;
        case 'r':               // running
                xt_iface_setflag(info, flags, invert, XT_IFACE_RUNNING, "running");
                return 1;
        case 'n':               // noarp
                xt_iface_setflag(info, flags, invert, XT_IFACE_NOARP, "noarp");
                return 1;
        case 'o':               // promisc
                xt_iface_setflag(info, flags, invert, XT_IFACE_PROMISC, "promisc");
                return 1;
        case 'm':               // multicast
                xt_iface_setflag(info, flags, invert, XT_IFACE_MULTICAST, "multicast");
                return 1;
        case 'd':               // dynamic
                xt_iface_setflag(info, flags, invert, XT_IFACE_DYNAMIC, "dynamic");
                return 1;
        case 'w':               // lower_up
                xt_iface_setflag(info, flags, invert, XT_IFACE_LOWER_UP, "lower_up");
                return 1;
        case 'a':               // dormant
                xt_iface_setflag(info, flags, invert, XT_IFACE_DORMANT, "dormant");
                return 1;
        default:
                return 0;
        }
}

static void xt_iface_match_final_check(unsigned int flags)
{
        DEBUGP("final_check...");
        if (!(flags & XT_IFACE_IFACE))
                exit_error(PARAMETER_PROBLEM,
                           "iface: You must specify an interface !");
        if ((flags == 0) || (flags == XT_IFACE_IFACE))
                exit_error(PARAMETER_PROBLEM,
                           "iface: You must specify at least one option !");
}

static void xt_iface_match_print(const void *ip,
                                 const struct xt_entry_match *match,
                                 int numeric)
{
        DEBUGP("print...");
        const struct xt_iface_match_info *info =
            (const void *) match->data;
        printf("--iface %s", info->ifname);
        xt_iface_print_option(info, XT_IFACE_UP, "up");
        xt_iface_print_option(info, XT_IFACE_BROADCAST, "broadcast");
        xt_iface_print_option(info, XT_IFACE_LOOPBACK, "loopback");
        xt_iface_print_option(info, XT_IFACE_POINTOPOINT, "pointopoint");
        xt_iface_print_option(info, XT_IFACE_RUNNING, "running");
        xt_iface_print_option(info, XT_IFACE_NOARP, "noarp");
        xt_iface_print_option(info, XT_IFACE_PROMISC, "promisc");
        xt_iface_print_option(info, XT_IFACE_MULTICAST, "multicast");
        xt_iface_print_option(info, XT_IFACE_DYNAMIC, "dynamic");
        xt_iface_print_option(info, XT_IFACE_LOWER_UP, "lower_up");
```

```
        xt_iface_print_option(info, XT_IFACE_DORMANT, "dormant");
        printf(" ");
}

static void xt_iface_match_save(const void *ip,
                                const struct xt_entry_match *match)
{
        DEBUGP("save...");
        xt_iface_match_print(ip, match, 0);
}

static struct xtables_match xt_iface_match_reg = {
        .name           = _MODULE_NAME,
        .revision       = _MODULE_REVISION,
        .version        = IPTABLES_VERSION,
        .family         = AF_INET,
        .size           = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .userspacesize  = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .help           = xt_iface_match_help,
        .init           = xt_iface_match_init,
        .parse          = xt_iface_match_parse,
        .final_check    = xt_iface_match_final_check,
        .print          = xt_iface_match_print,
        .save           = xt_iface_match_save,
        .extra_opts     = xt_iface_match_opts,
};

static struct xtables_match xt_iface_match6_reg = {
        .name           = _MODULE_NAME,
        .revision       = _MODULE_REVISION,
        .version        = IPTABLES_VERSION,
        .family         = AF_INET6,
        .size           = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .userspacesize  = XT_ALIGN(sizeof(struct xt_iface_match_info)),
        .help           = xt_iface_match_help,
        .init           = xt_iface_match_init,
        .parse          = xt_iface_match_parse,
        .final_check    = xt_iface_match_final_check,
        .print          = xt_iface_match_print,
        .save           = xt_iface_match_save,
        .extra_opts     = xt_iface_match_opts,
};

void _init(void)
{
        DEBUGP("_init...");
        xtables_register_match(&xt_iface_match_reg);
        xtables_register_match(&xt_iface_match6_reg);
}
```

9. IRODALOMJEGYZÉK

[1] „ECN - RFC2481”
1999. január.
http://www.faqs.org/rfcs/rfc2481.html

[2] „Linuxvilág #42” *Linuxvilág.hu.*
2004. július.
http://www.linuxvilag.hu/content/files/cikk/42/cikk_42_25_27.pdf

[3] „Patch-O-Matic-NG” *Netfilter.org.*
http://www.netfilter.org/projects/patch-o-matic/index.html

[4] „RFC 2863 - The Interfaces Group MIB” *Internet FAQ Archives.*
2000. június.
http://www.faqs.org/rfcs/rfc2863.html

[5] „The History of Linux.” *Linux Online.*
http://www.linux.org/info/linux_timeline.html

[6] „O’REILLY Linux iptables zsebkönyv” *Gregor N. Purdy*
Kiskapu Kft, 2006
ISBN: 9789639637122

[7] „Programozás II.” *Bauer Péter – Hatwágner F. Miklós*
Széchenyi István Egyetem, 2006.
http://jegyzet.sze.hu/letolt.php?dwn=1programozasii