TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Vladislav Alenitsev

Failisüsteemi hägutestimise raamistik

Bakalaureusetöö (9 EAP)

Juhendaja: Meelis Roos
Juhendaja: Kristjan Krips

Tartu 2017

Failisüsteemi hägutestimise raamistik

Lühikokkuvõte:
Tarkvara testimine muutub järjest tähtsamaks seoses koodi keerukuse kasvuga. Isegi asjatundjate poolt koostatud rakendused võivad sisalda vigu juhul, kui tegemist on keskmisest keerukamate ja mahukamate programmidega. Tarkvarast vigade otsimine on raske ja ajakulukas ülesanne, kuid seda tööd saab testimisvahendite abil lihtsustada. Käesoleva töö eesmärgiks oli ehitada raamistik, millega saab hägutestimise abil otsida Linuxi failisüsteemides vaheletrügimisnõrkusi.

Koostatud raamistik erineb olemasolevatest lahendustest selle poolest, et testimist sooritatakse tavapärasest kihist allpool, failisüsteemi tasemel. Töö tulemuseks on paindlik testimisraamistik, mis lahendab andmevahetuse probleemi ning lihtsustab erinevate häguritüüpide implementeerimist. Vaatamata sellele, et arenduses keskenduti Ext4 failisüsteemile, on pandud alus võimalikule edasisele arendusele.

Võtmesõnad:
TOCTOU, vaheletrügimisnõrkus, hägutestimine, hägur, failisüsteem

CERCS: P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Filesystem fuzz testing framework

Abstract:
The need for software testing increases in proportion to the software's cyclomatic complexity. Even expert programmers might make mistakes while developing complex software. Discovering those mistakes is a difficult and time consuming task which can be simplified through different instruments. In the present thesis a fuzz testing framework was built, which can be used for finding time-of-check-to-time-of-use type bugs in Linux filesystems.

The main difference between this framework and already existing fuzzing tools, is that this framework executes its testing on filesystem level. As a result of this work, a flexible framework was constructed, which solves the data transfer problem and simplifies the implementation of different types of fuzzers. Despite the fact that the development was focused on Ext4 filesystem, it is possible to extend the tool to include other filesystems.

Keywords:
TOCTOU, time-of-check-to-time-of-use bug, fuzz testing, fuzzer, filesystem

CERCS: P170, Computer science, numerical analysis, systems, control

Sisukord

1 Sissejuhatus

Iga päev arendatakse tarkvara, mille eesmärgid varieeruvad klahvivajutuse töötlemisest kuni raketi juhtimiseni. Juhul kui tarkvara koodibaas muutub suureks, siis võivad isegi parimad tarkvaraarendajad teha vigu, sest koodiridade kasvuga suureneb enamasti ka tarkvara keerukus. Steve McConnell, tarkvaraarenduse ja projektijuhtimise ekspert, väidab oma teoses "Code Complete", et keskmiselt tehakse 1 kuni 25 viga tuhande koodirea kohta [McC04, 558]. Seetõttu võivad tarkvara keerukusest tulenevad vead jääda märkamatuks ka tarkvara testimise käigus.

Operatsioonisüsteemi turvalisuse verifitseerimine on keeruline ülesanne ning seda on tehtud vaid mõne väikse operatsioonisüsteemiga [BBB10]. Kuna Linuxi kerneli, operatsioonisüsteemi tuuma, lähtekood on avalik, siis oleks selle formaalne verifitseerimine teoreetiliselt võimalik, aga praktiliselt oleks vastav ülesanne äärmiselt keerukas, sest Linuxi kernel koosneb 20 miljonist koodireast [Lee15]. Vastavalt ülaltoodud McConnelli väitele ja Linuxi kerneli koodiridade arvule võib eeldada, et ka Linuxi kernelis leidub vigu. Seda illustreerib statistika, mille kohaselt on 2017. aasta jaanuarist aprillini avaldatud 266 kerneli turvaprobleemi, mis osutub kümne viimase terve aasta kohta suurimaks turvaaukude koguseks. Võrdluseks, 2016. aasta vältel avaldati kokku 217 haavatavust [MIT17].

Enamasti kontrollivad operatsioonisüsteemid faili staatust enne kui alustatakse faili mõjutavaid tegevusi. Näiteks Unix ja Windows üritavad seda kontrollida enne, kui lubatakse faili muuta. Selle tegevuse eest vastutab failisüsteem. Kusjuures, oma arhitektuurilise ehituse poolest eeldab faili haldav programm, et pärast staatuse kontrolli pole andmed kettal muutunud ning samuti eeldatakse, et failisüsteem töötab korrektselt. Ajahetkel, kus faili staatuse kontroll on lõppenud ning faili muutmise operatsioon pole veel alanud, on võimalik muuta kontrollimises kasutatud parameetreid, mis võib põhjustada vale operatsiooni rakendamist failile. Taolist tegevust nimetatakse TOCTOU (*time of check to time of use*) tüüpi ründeks.

TOCTOU ründe omapära tõttu on raske leida selle eest kaitset. Seetõttu on ka paljud failisüsteemid ohustatud, kuna vastava ründe vältimine on keerukas ülesanne. Ründe tagajärjed võivad olla väga erinevad, näiteks võib ründaja saada juurdepääsu andmetele, mis poleksid teistmoodi kättesaadavad. Nii võib ründaja saada lugemis- ja/või kirjutamisõigused failile, mida ta muidu poleks võimeline muutma või lugema [MIT]. Samuti võib ründe tulemuseks olla süsteemi krahh [Lev15].

Käesoleva töö eesmärgiks on ehitada raamistik, mis võimaldaks sooritada hägutestimist (*fuzz testing*) erinevate failisüsteemide peal. Niisuguse raamistiku abil oleks võimalik otsida failisüsteemist turvaauke, mis põhinevad TOCTOU ründel. Taolisi turvaprobleе-

me on avastatud näiteks Linuxi failisüsteemi Ext4 implementatsioonidest kuni kerneli versioonini 4.5. [Nem16].

Töö esimeses peatükis antakse ülevaade töö taustast ning seletatakse lahti kõige tähtsamad terminid ja mõisted. Seletatakse, miks need on selles töös vajalikud ning kuidas neid kasutati. Teises peatükis esitatakse praktilise osa detailne kirjeldus, antakse ülevaade arenduse käigust ning tuuakse näiteid raamistiku kasutusest. Samuti käsitletakse arenduse käigus tekkinud probleeme ning seda, kuidas need lahendatud said. Kolmandas peatükis kirjeldatakse saadud tulemust, võrreldakse seda olemasolevate raamistikega ning arutatakse edasise töö võimaluste üle.

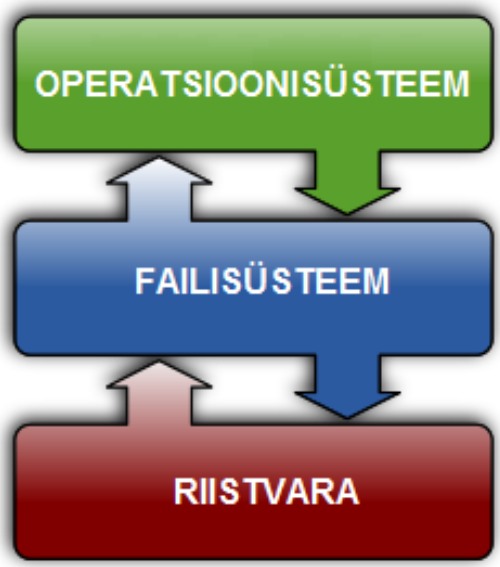
2 Taust

2.1 Eesmärk

Antud peatüki eesmärk on anda lugejale taustinformatsiooni ning selgitada, miks on failisüsteem operatsioonisüsteemile vajalik. Antakse ülevaade Ext4 failisüsteemist ning selle struktuurist. Seletatakse, mis on hägutestimine ja TOCTOU rünne, räägitakse efektiivse ründe tagajärgedest ning teistest töö jaoks olulistest mõistetest.

2.2 Failisüsteem

Failisüsteem on andmestruktuuride, algoritmide ja tarkvara kogum, mille alusel toimub andmete organiseerimine, hoiustamine ja nimetamine arvuti või muu elektroonse seadme infokandjal. Piltlikult öeldes operatsioonisüsteemid või muud programmid teostavad andmevahetust riistvaraga läbi failisüsteemi (joonis 2.1). Samuti määrab failisüsteem erinevaid parameetreid nagu näiteks maksimaalne faili suurus ja faili või kataloogi nimetuse suurus. Failisüsteemita ei oleks võimalik eristada, kus lõpeb andmekandjal üks andmetükk, ning kus algab teine. On loodud hulgaliselt erinevaid failisüsteeme, millest igaühel on erinev ülesehitus ning eesmärk [Wik17c]. Linuxi jaoks on laialdasemalt kasutuses failisüsteemid Ext2, Ext3, Ext4, XFS ja JFS. Windowsi korral on populaarsemad FAT, exFAT, NTFS ja ReFS. Selle töö raames käsitletakse Linux operatsioonisüsteemi ning selle jaoks loodud failisüsteemil Ext4. Seetõttu annab järgmine jaotis ülevaate failisüsteemist Ext4.



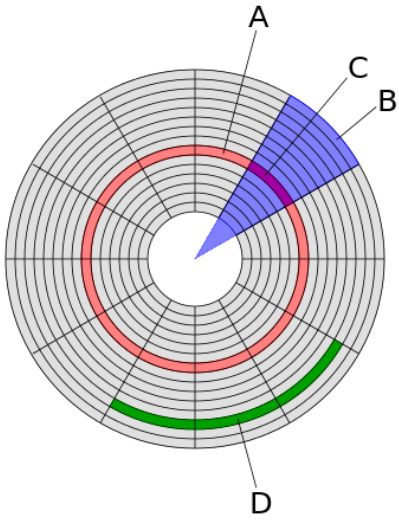
Joonis 2.1. *Operatsioonisüsteem loeb ja kirjutab riistvarale andmeid läbi failisüsteemi* [Gol10].

2.3 Ext4 failisüsteem

Ext4 (*fourth extended filesystem*) võeti kasutusele 2008 aastal Linuxi kerneli versioonis 2.6.28 ning see sisaldub igas tänapäevases distributsioonis. Ext4 on Ext3 edasiarendus, see on oma eelkäijaga ühilduv, mis tähendab seda, et ketast formaadiga Ext3 saab haakida kui Ext4 ilma, et peaks midagi ketta peal kohandama. Ext4 töökindlus ning

maksimaalsed failisüsteemi ja faili andmemahtuvused on suuremad kui tema eelkäijal [wik17b].

Failisüsteem jagab andmekandjal oleva ruumi ühikuteks, mida nimetatakse plokkideks (*block*). Plokk on sektorite grupp, mille suurus on vahemikus 1 KiB kuni 64 KiB, kusjuures sektorite arv plokis peab olema kahe aste. Antud kontekstis, sektor (joonis 2.2 C) on ala, mis jääb ketta geomeetrilise sektori (joonis 2.2 B) ning ketta raja (joonis 2.2 A) ühisosssa. Tavaliselt on Ext4 failisüsteemi ploki suurus 4 KiB. Kõik plokid, mida on võimalik andmekandjalt lugeda on jagatud plokirühmadeks (*block groups*). Üks plokirühm mahutab endas 32 768 ploki, mille tavaline pikkus on vastavalt 128 MiB. Jõudluse mõttes paigutab failisüsteem ühte rühma need plokid, mis on üksteisele võimalikult lähedal. Kuna ülaltoodud ühikud ei võrdu tavapärasemate andmemahtuvuse ühikutega nagu kB, MB jne, siis on mõtet tähelepanu pöörata nende vastavustele, mis on toodud tabelis 2.1.



Joonis 2.2. *Ketta struktuur, A rada (track), B geomeetriline sektor, C raja sektor (track sector), D klaster (cluster) [oIS13].*

Tabel 2.1. *Andmemahu ühikute vastavus. MB megabyte, kB kilobyte, KiB kibibyte, MiB mebibyte.*

	KiB suurus	**kB suurus**	**MiB suurus**	**MB suurus**
1 MB	976.563	1000	0.95367	1
1 MiB	1024	1048.58	1	1.04858

Tabel 2.2. *Ext4 failisüsteemi maksimumid 32-bit ja 64-bit korral [wik17a].*

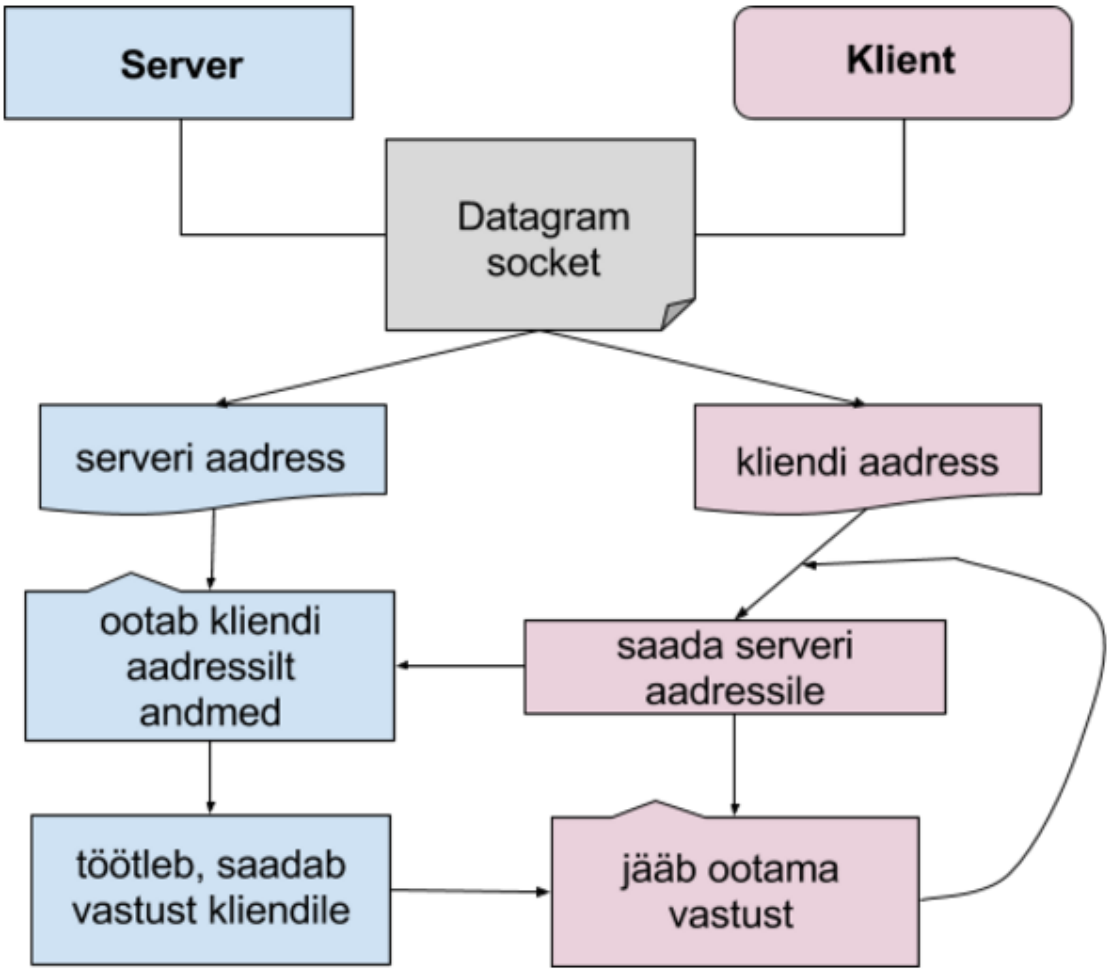
32-bit			
ploki suurus	1 KiB	4 KiB	64 KiB
plokkide arv	232	232	232
plokke rühmas	8 192	32 768	524 288
plokirühma suurus	8 MiB	128 MiB	32 GiB
64-bit			
ploki suurus	1 KiB	4 KiB	64 KiB
plokkide arv	264	264	264
plokke rühmas	8 1923	32 768	524 288
plokirühma suurus	8 MiB	128 MiB	32 GiB

Tabelist 2.2 saab näha, millised maksimumid on seatud Ext4 väljadele. 32-bit ja 64-bit süsteemide korral erinevad on ainult maksimaalsed plokkide arvud. Märkimisväärne on Ext4 failisüsteemi maksimaalne suurus, mis on 1 EB ehk 1 eksabait ehk 10^9 gigabaiti. Ning ühe faili maksimaalne suurus on 16 TB, mis on 8 korda suurem võrreldes Ext3. Kõik Ext4 failisüsteemi väljad on ketta peale kirjutatud *little-endian* järjestuses [wik17a]. Sellest, mida see tähendab ja miks see on oluline, räägitakse jaotises 2.5.

2.4 Datagrammsokkel

Datagrammsokkel (*datagram socket*) on protsessidevahelise suhtluse (*interprocess communications socket*) või võrgu sokkel (*network socket*). Kuna selles töös võrgu soklit vaja ei lähe, siis piisab vaid üldisest teadmisest, et selline sokli tüüp on olemas. Protsessidevahelise suhtluse sokkel on lõpp-punkt, mille kaudu kaks või enam ühes süsteemis tegutsevat protsessi saavad vahendada andmeid.

Joonisel 2.3 on kujutatud datagrammsokli tööpõhimõte. Üks protsess tegutseb serverina ning tekitab datagram socket tüüpi soklifaili ja jääb ootama kliendi poolt saadetud andmepaketti. Teine protsess tegutseb kliendina, ühendub soklifailiga ning saadab vajalikud andmed serverile. Server võtab vastu kliendi poolelt saadetud andmed, rakendab vajalikud muudatused ning saadab kliendi aadressile vastuse. Sellise soklitüübi korral on tähtis, et oleks määratud andmete vastuvõtja aadress.



Joonis 2.3. *Datagram socketi suhtluse põhimõte.*

Selles töös kasutati datagrammsoklit selleks, et konstrueerida andmevahetust hägutestimise (vt. jaotis 2.6) ning UML ehk User-mode Linuxi (vt. jaotis 2.8) programmiosade vahel. Raamistiku hägutestimise osa tegutseb serverina ning UML kliendina. Kliendi poolt saadetakse nihe (*offset*), puhvri pikkus ning puhver (*buffer*). Nihe on kaugus plokkseadme algusest vajaliku ploki andmeteni. Puhver on mäluosa, mida kasutatakse andmete ajutiseks salvestamiseks, selleks, et protsessor saaks neid andmeid kiiresti lugeda ning saata edasi andmekandjale. Puhvri pikkus kirjeldab baitide arvu puhvris.

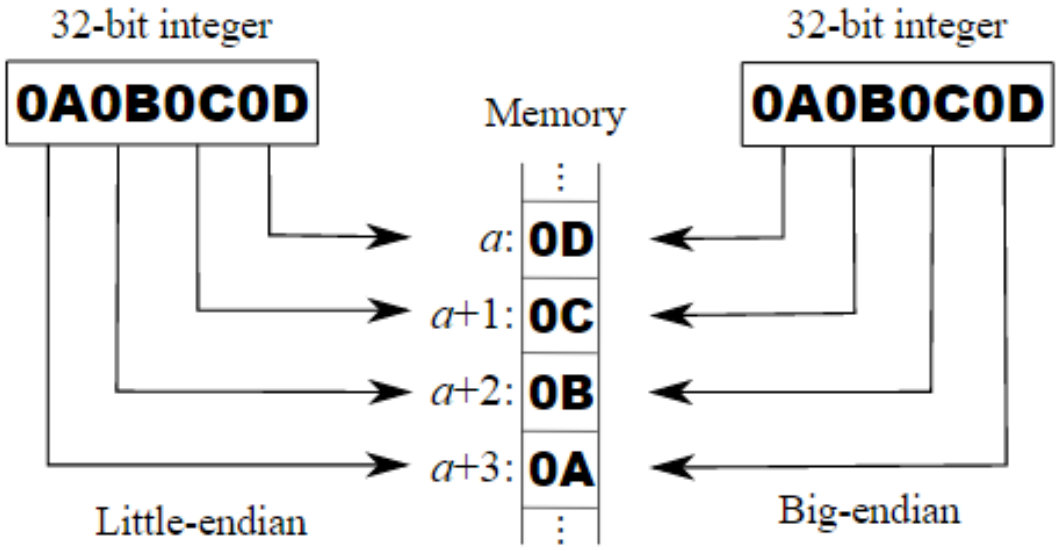
2.5 Baidijärjestus

Kuna selles töös on suhteliselt palju tegelemist baidijadade protsessidevahelise edasitagasi saatmisega ning, et raamistiku sisetöö käigus ei toimuks andmete kontrollimata

muutmist, siis on vaja teada sellist terminist nagu baidijärjestus (*endianness*). On olemas kaks põhilist liiki järjestust:

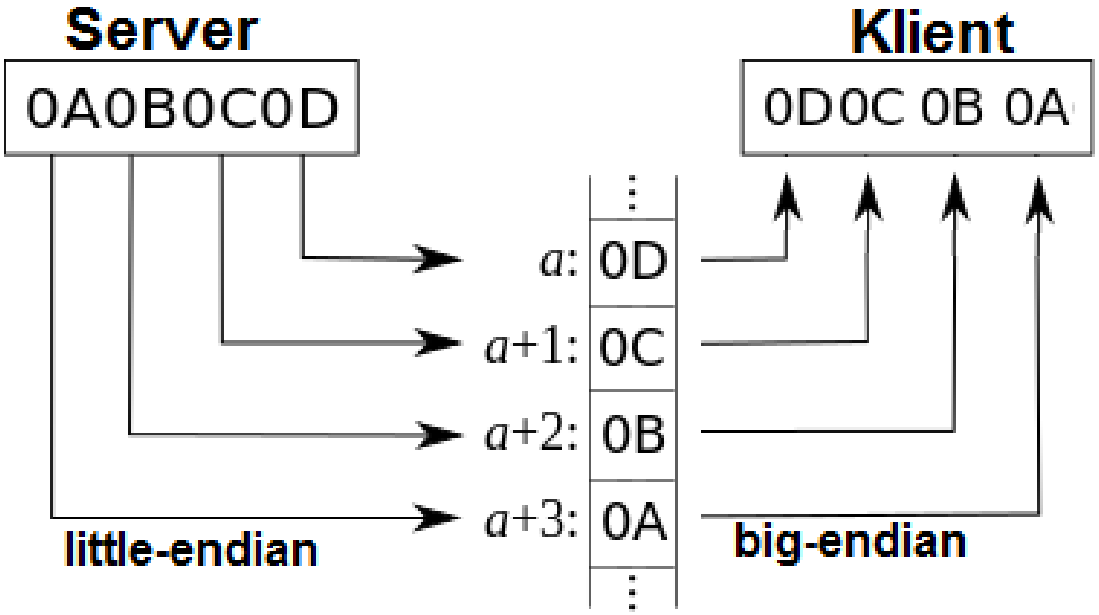
- pöördjärjestus (*little-endian*);
- tavajärjestus (*big-endian*).

Little-endian korral järjestatakse baidid mällu sellisel viisil, et vähima tähtsusega bait saab väikseima aadressi. *Big-Endian* korral väikseim aadress läheb kõige tähtsamale baidile [UMD]. Suurima tähtsusega bait ehk MSB (*most significant byte*) tähendab sellist baiti, mis mingisuguse andmetüki baidilises kujus omab suurimat väärtust. Näiteks, kümnendsüsteemi arv 168496141 on kuueteistkümnendsüsteemis esitatud kujul 0A0B0C0D. See on 32-bitine ehk 4 baidine baidijada, kus baidid on vastavalt 0A; 0B jne. Tema tähtsaim bait on 0A. Baidijada paigutus mällu näeks välja nii nagu on kujutatud alumisel joonisel 2.4.



Joonis 2.4. *Little-endian ja big-endian baidi paigutus [Sha07a, Sha07b].*

Kui süsteem, mis vaikimisi kasutab *little-endian*'i, dekodeerib baidijada, mis on kokku pandud kasutades *big-endian*'it, siis saadud tulemus ei võrdu esialgse, saadetava tulemusega. Eelmises peatükis anti seletus datagrammsokkli tööpõhimõtele. Selleks, et serveri poolelt saadetavad andmed oleksid õiged nii kliendi poolel ja vastupidi, peab olema määratud õige baidi järjestuse tüüp, muidu võib juhtuda olukord, mis on kujutatud joonisel 2.5.



Joonis 2.5. *Little-endian järjestus pannakse kokku big-endianiga.*

Jooniselt 2.5 on näha, et kliendi poolel on kuvatud kuueteistkümnendsüsteemi arv 0D0C0B0A, mille kümnendüsteemi vaste 218893066 ei võrdu serveripoolel kujutatud väärtusega.

2.6 Hägutestimine

Hägutestimine (*fuzz testing*, *fuzzing*) on tarkvara testimise meetod, kus testitavale tarkvarale antakse sisendiks modifitseeritud, valed või juhuslikud andmed ning seejärel jälgitakse programmi tegevust. Lisaks turvaaukudele võib tuvastada ka programmis olevaid loogikavigu, mis ei tekita turvaprobleeme. Testimismeetodi nimest tuleb ka nimetus programmi jaoks, mis jooksutab taolisi teste – hägur (*fuzzer*).

Fuzzing võib olla strukturaaltestimise (*white box testing*), funktsionaaltestimise (*black box testing*) või nende kombinatsiooni (*grey box testing*) alammetoodika. Enamasti ehitatakse *fuzzer* siiski kas *black box* või *grey box* meetodit silmas pidades [SGA07, 17]. Strukturaaltestimine on testimismeetod, mis eeldab programmi lähtekoodi olemasolu. Seega, kui vastav teave on olemas, siis on võimalik kasutada *white box fuzzer*'it, mis häguandmete koostamisel arvestaks programmi põhimõtega. Funktsionaaltestimine tähendab seda, et puuduvad andmed testitava programmi ülesehituse kohta. On teada vaid see, milline peaks olema väljund. *Grey box testing* on testimismeetod, milles kasutatakse koodi pöördkonstrueerimist (*reverse engineering*) selleks, et saavutada mingisugune

ettekujutus programmi lähtekoodist. Tabelis 2.3 on välja toodud iga metoodika kohta nende eelised ja puudujäägid seoses haavatavuste otsimisega.

Tabel 2.3. *Testimismeetodite eelised ja puudujäägid [SGA07, 17-31].*

Metoodika	**Eelised ja puudujäägid**
White box	On võimalik saavutada täielik koodikate (code coverage), kuna programmi lähtekood on teada. Koodikeerukuse (code complexity) tõttu on võimalik koostada valepositiivseid tulemusi, mis ei kindlusta programmi korrektsust. White box testimist ei ole võimalik sooritada, kui lähtekoodi pole teada.
Black box	Funktsionaaltestimist on alati võimalik korraldada ning see saab olla kasulik isegi siis, kui ei ole lähtekoodi või teavet programmi tööpõhimõtte kohta. Vigast olukorda on võimalik lihtsalt reprodutseerida, kui viga põhjustav sisend on teada. Testimismetoodika on äärmiselt lihtne ning ei nõua erilisi teadmisi. Selle metoodika üks põhilisi puudujääke on koodikate (*code coverage*), kuna ei ole võimalik teada saada, millisel määral on programm testitud ning kas sooritatud testimisest piisab. Vaatamata sellele, et funktsionaaltestimine ei nõua suuri teadmisi programmeerimisest, on turvaaukude leidmine raskendatud, kui rünne koosneb mitmest osast. Näiteks olukorras kus üks sisend on mõeldud selleks, et programmi haavata ning teine kasutab seda haavatavust ära, siis taoline programmiviga nõuab mingit teadmist programmi tööpõhimõtte kohta ning jääb tavaliselt avastamata.
Grey box	Reeglina on tarkvara binaarsed versioonid saadaval, mis võimaldavad lähtekoodi mingil määral pöördkonstrueerida, erandiks on veebiteenused ja -rakendused. Koodi pöördkonstrueerimine on erialane oskus, mis ei võimalda igaühel teostada antud testimismetoodikat.

Vaatamata väljavalitud metoodikale on fuzzimise etapid alati samad [SGA07, 39-40]:

1. selgitada välja testitav programm;
2. selgitada välja sisendi formaat või andmetüüp;
3. genereerida häguandmed;
4. sisestada genereeritud andmed;

5. jälgida programmi käitumist ning selle väljundit;

6. teha otsus turvaaugu või probleemide olemasolu kohta.

Häguandmete genereerimiseks on enamasti kaks võimalust. Esimene on sisendi muteerimisel baseeruv (*mutation-based*) andmete genereerimine, mille kohaselt iga testimise iteratsiooni korral muudetakse olemasolevaid sisendandmeid. Teiseks võimaluseks on andmete juhuslik genereerimine (*generation-based*). Sellisel viisil koostatakse sisendit algusest peale pidades silmas testitava programmi sisendi andmetüüpi [SGA07, 45]. Lisaks kõikidele ülaltoodud häguri klassifikaatoritele eristatakse veel nelja tüüpi hägureid, mis on välja toodud tabelis 2.4.

Tabel 2.4. *Häguri tüübid [Gui10].*

Tüüp	**Kirjeldus**
Juhuslik hägur	Antud tüüpi hägur koostab täiesti juhuslikud andmed ning söödab need rakendusse. See on kõige primitiivsem hägutestimise viis.
Muteeriv hägur	Taoline hägur kogub rakenduse näidissisendeid ning muundab neid. Näiteks, rakenduse korral, mille sisendiks on baidijada, võib muteeriv hägur saadud baidijada järjestust muuta või asendada selle täiesti tühja, kuid sama pikkusega baidijadaga.
Genereeriv hägur	Hägur modelleerib rakenduse jaoks spetsiaalset sisendit, mis testib programmi haavatavamaid osi. Sellist tüüpi hägurit on raske ning ajakulukas implementeerida, kuid reeglina annab see parima tulemuse.
Evolutsiooniline hägur	Sellist tüüpi hägur on genereeriva häguri edasiarendus, mis modifitseerib sisendid vastavalt koodikattele. Tegu on kõige targema häguritüübiga.

Lisaks nendele puudujääkidele, mis tulenevad häguri tüübist, leiduvad ka üldisemad hägutestimisega seotud probleemid [Lev15]:

- juhusliku häguri korral ei saa sisendi genereerimisel olla täiesti kindel selles, et kõik sisendi variatsioonid on läbi proovitud. Seega puuduvad testimise läbimist tähistavad kriteeriumid;

- hägutestimine on aeganõudev protsess, mis tähendab seda, et vea leidmiseks ei piisa ühekordsest häguri käivitamisest, vaid tuleb pidevalt testida;

- avastatud vigu on tavaliselt raske käsitsi taastekitada.

2.7 TOCTOU

TOCTOU (*time of check to time of use*) või vaheletrügimisnõrkus on tarkvara programmivea (*software bug*) tüüp. Sellise veatüübi korral on ründajal võimalik muuta süsteemset objekti (nt. faili), pärast seda kui objekti staatust on kontrollitud ja enne kui vastav operatsioon on objektile rakendatud. Objekti staatuse kontroll võib olla näiteks objekti juurdepääsuõiguste või lukustatud oleku kontroll. TOCTOU ründe jaoks on vaja kahte sündmust [WP05]:

1. süsteem teostab ressurssi staatuse kontrolli;
2. süsteem rakendab ressursile mingit operatsiooni, eeldusel, et ressursi staatus ei ole muutunud.

Eduka ründe tavalisemad tagajärjed:

- ründaja saab juurdepääsu ressursile;
- ründaja saab lugemis- ja/või kirjutamisõigused teatud failidele;
- ründaja saab kustutada faile, milledele tal muidu puuduks juurdepääs [MIT];
- saavutatakse süsteemi krahh [NC16].

Ressursi all mõeldakse siin süsteemi objekti (faili, mälu osa, vms), mis on modifitseeritav TOCTOU rünnet teostava programmi poolt.

Probleemi illustreerimiseks on joonises 2.6 toodud TOCTOU näide rakenduse tasemel:

```
1 if(!access(file,W_OK)) {
2     f = fopen(file,"w+");
3     operate(f);
4     ...
5 }
6 else {
7     fprintf(stderr,"Unable to open file %s.\n",file);
8 }
```

Joonis 2.6. *TOCTOU näidis [MIT].*

Kõigepealt kontrollitakse *if* ploki tingimuses, *access()* meetodi abil, kas kasutaja saab tavaõigustega failiga midagi teha. Juhul, kui see on võimalik, siis avatakse fail muutujasse *f* ning sooritatakse selle peal edasised operatsioonid. Juhul, kui kasutajal puuduvad õigused faili käsitlemiseks, siis läheb programm *else* harru.

Kui ülaltoodud C kood on jooksutatud *setuid* programmis, siis on taolises programmis TOCTOU viga. *Setuid* võimaldab programmi käivitajal kasutada kõrgemaid õigusi, kui

tal tegelikult on. Näiteks saab selle abil külaliskonto käivitada programmi juurkasutaja õigustes. Eelnevas näites olev funktsioon *access()* kontrollib, kas kasutaja saaks failiga opereerida isegi siis, kui setuid lippu ei oleks. Antud koodi probleem seisneb selles, et seal pole mingit kontrolli selle üle, kas failimuutuja *f* on muutunud pärast funktsiooni *access()* väljakutset. Ründaja saab nimeviidaga (*symlink*) viidata teisele failile, millele tal muidu ei oleks õigusi ligi pääseda, ning programm on siis võimeline sooritama ükskõik milliseid operatsioone ümbersuunatud faili peal. Nimeviit on Unix'i fail, mis viitab teisele failile. Seda saab luua käsuga *ln -s lähtefail uus*. Seega TOCTOU ründe abil on teatud tingimustel võimalik saavutada õiguste vallutust (*privilege escalation*). Eduka TOCTOU ründe jaoks ei ole tingimata vaja seda, et programmi käivitaja oleks juurkasutaja õigustega. Kui rakendus saab teha sellist operatsiooni, mille rakenduse kasutajal puuduvad õigused, siis rakendus on ründe poolt haavatav.

Faili lukustamine (*file locking*) on mehhanism, mis lubab ainult ühel kasutajal või protsessil saavutada juurdepääsu lukustavale failile. See on tavaliseim TOCTOU vastane meetod, kuid faili lukustamine ei laiene failisüsteemi nimeruumile (*namespace*) ega metaandmetele. Nimeruum on mingi kindla kokkulepe alusel koostatud nimede rühm. Kõigil ühte nimeruumi kuuluvatel objektidel (nt. faili asukohal, aadressil) on unikaalne nimetus. Metaandmed on andmed, mis kirjeldavad andmeid. Antud töös koostatakse raamistik, mis võimaldab otsida TOCTOU ründe võimalusi failisüsteemi draiverite realisatsiooni tasemel, kus ei tegeleta failipuu objektidega.

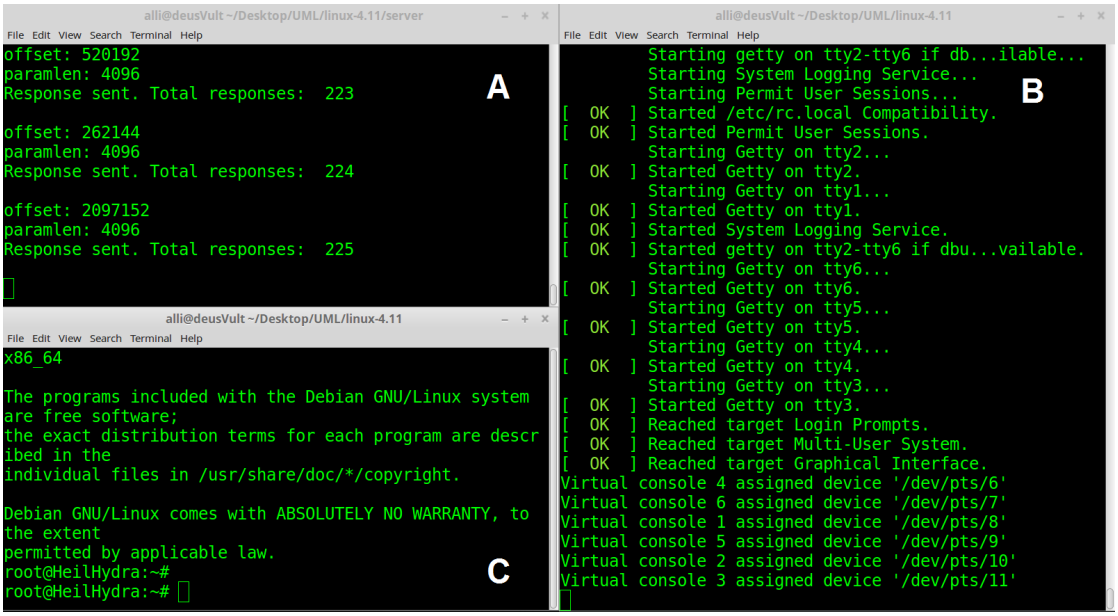
2004. aastal avaldati uurimistöö, mille alusel Linuxi failisüsteemides ei ole olemas kindlat viisi TOCTOU tüüpi rünnete ärahoidmiseks [Ass04]. Pärast uurimuse avalikustamist pakuti lahenduseks välja transaktsioonide lisamist failisüsteemi või operatsioonisüsteemi kernelisse. Transaktsioonid võimaldavad teostada kokkulangevuse kontrolli (*concurrency control*), mis omakorda kindlustab õige operatsioonitulemuse mitme samaaegselt jooksva protsessi korral. Sellist metoodikat silmas pidades on arendatud välja TxOS kernel [PHR+09] ning Linuxi failisüsteem Valor [SGC+09], kuid teistes Linuxi failisüsteemides seda tehnoloogiat implementeeritud ei ole.

2.8 User-mode Linux

User-mode Linux ehk UML on virtuaalne masin, mida võib käivitada tavalise protsessina mõne muu Linux operatsioonisüsteemi sees. UML-i seadistus on paindlik ning see võimaldab määrata rohkem riist- ja tarkvara ning virtuaalseid ressursse kui UML-i jooksutavas masinas tegelikult leidub. See võimaldab anda ligipääsu ainult vajalikule riistvarale. Lisaks on oluline, et kogu virtuaalse ketta sisu paikneb UML-i jooksutavas masinas ühes failis. Tavaliselt kasutatakse UML-i virtuaalse serveri hoidmiseks, kerneli arenduseks, õppe eesmärgil ning ka erinevate rakenduste turvaliseks käivitamiseks [Dik]. Antud töös korraldatud arendus toimus just UML-il, mis kasutas kerneli ver-

siooni 4.11, sest UML-s on suur virtuaalsete plokkseadmete emuleerimise kiirus, mis võimaldas teha kiiremaid katseid.

Joonisel 2.7 on kujutatud hosti sees käivitatud UML-i protsess (konsool B) ning seda protsessi haldav konsooliaken (konsool C). Vasakul ülemises nurgas avatud konsooliaknas (konsool A) on serverina käituv programm (vt jaotis 3.3.1).



Joonis 2.7. *Käivitatud UML ja teda kuulav server host masinas.*

Selleks, et UML-i koodimuudatused jõustuksid järgmisel käivitamisel, tuleb anda UML-i kaustas avatud terminalis *make* käsk failide kompileerimiseks. Käsk jooksutab *makefile*'i. See on fail, kus on kirja pandud kõik teised vajalikud käsud failide kompileerimiseks. Tavaliselt programm koosneb paljudest failidest, mida käsitsi kompileerida oleks ajakulukas. *Make* käsk võimaldab seda protsessi automatiseerida. Käsul on hulgaliselt lippe (*flag*), kuid peamised kaks, mida kasutati arenduses olid järgmised [Fou16]:

- *make -j4 ARCH=um* käsk kompileerib UML-i faile. Lipp *-j* tähendab, et käsk tohib luua ülimalt ära märgitud arvu lõime (*thread*) käsu sooritamiseks. Selle lipu analoog on *–jobs*. Neljatuumalise protsessi korral määratakse vastav arv lõimesid. Sellise käsuga saavutatakse optimaalseimat protsessiressursi kasutust.

- *make V=1 ARCH=um* käsk lipuga *V=1*, mis lisaks UML-i failide kompileerimisele toob konsooliaknas välja *makefile* käsud. Kasutaja saab näha, millised käsud kutsutakse välja ning milliste käskude jooksutamine lõpeb veaga. Vastava lipu analoog on *VERBOSE=1*.

Mõlema *make* käsu puhul on kasutusel *ARCH=um*, mis ütleb host masina kernelile, et kompileerida tuleb UML-i, millel on oma arhitektuur. Kui see jääb märkimata, siis kompileeritakse hosti kerneli arhitektuuriga [Dik]. UML-i käivitamiseks tuleb UML-i kataloogi konsoolis sisestada järgmine käsk:

```
./linux mem=256M ubd0=[virtuaalketta 1 asukoht] root=/dev/ubda umid=
    testmasin ubd1=[virtuaalketa 2 asukoht]
```

Käsu sisestamisel on vajalik seadistada lipud *ubd0=[virtuaalketta 1 asukoht]* ning *ubd1=[virtuaalketta 2 asukoht]*. Kandiliste sulgude asemel tuleb märkida failitee kettatõmmistele, mida UML hakab kasutama. Kaks ketast määratakse sellel põhjusel, et üks oleks puutumatu põhiketas, mille peal UML-i käivitatakse ning teist hakatakse kasutama hägutestimisel. Õnnestunud UML-i käivitamisel (joonis 2.7 B) ilmuvad konsooli aknasse sõnumid:

```
Virtual console X assigned device '/dev/pts/Y'
```

Ülaltoodid väljundis on X ja Y on mingid numbrid. Lisaks läheb vaja programmi, millega saaks UML-i protsessiga ühenduda läbi kõrvalise terminali. Arenduse käigus kasutati programmi Screen. Seda on võimalik Linux Mint peale installida järgmiste käskudega:

```
sudo apt-get update
sudo apt-get install screen
```

Nüüd saab uues konsooliaknas sisestada käsu

```
screen /dev/pts/Y
```

mille peale käivitub UML-i konsool. Kasutajanimi ja parool on “root” (ilma “”). Peale nende sisestust ilmub konsooli sõnum, mis on kujutatud joonisel 2.7 aknas C. UML-i töö korrektseks peatamiseks tuleb samasse konsooli sisestada käsk *halt*.

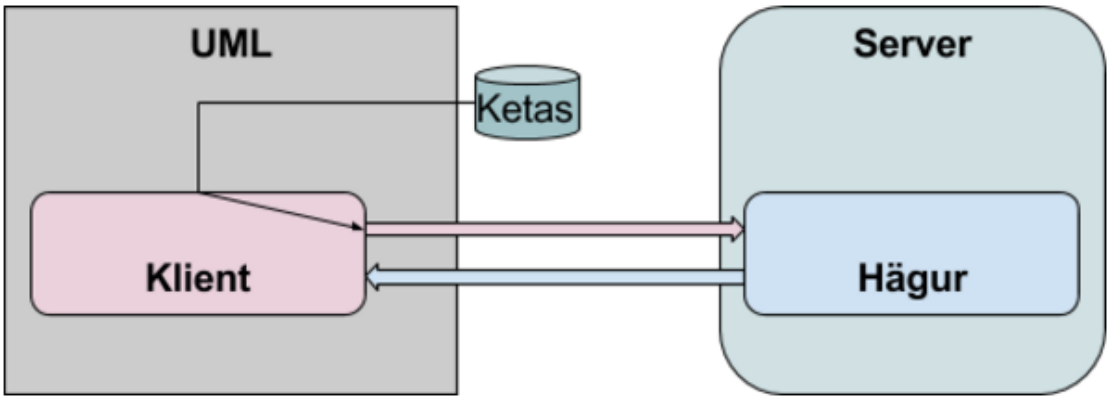
3 Raamistiku ülesehitus

3.1 Eesmärk

Antud peatükis antakse ülevaade nii arendamise protsessist kui ka valminud rakendusest. Seletatakse, millistest osadest raamistik koosneb ning kuidas toimub selle töö. Samuti räägitakse sellest, kuidas kõik osad sisse lülitada ning milline näeb välja õnnestunud raamistiku käitamine. Antud peatükk on vajalik selleks, et teada saada, kuidas antud töö raames koostatud raamistikku kasutada. Kogu informatsioon arenduse käigus kasutatud programmi, kirjutatud koodi ja selle paiknemise kohta ning põhjalik juhend raamistiku üles seadmise kohta on viidatud lisas Lisa A: Skriptid.

3.2 Raamistiku tööpõhimõte

Töö eesmärgiks on luua raamistik, kus oleks võimalik implementeerida erinevaid hägureid ning keskenduda testimisele. Joonisel 3.1 on kujutatud UML-i ja serveri andmevahetus. Raamistiku võib põhimõtteliselt jagada kaheks osaks. Esimene osa on UML-i funktsionaalsus, mis vastutab ketta pealt andmete lugemise eest. Teine osa on server, kus saaks implementeerida erinevaid hägureid loetud andmete muutmiseks. Arenduse alguses tehti otsus konstrueerida raamistiku teine, serveri osa Pythonis. Suurem osa UML-st, nagu ka Linuxist, on kirjutatud C keeles ning erinevalt Pythonist, ei ole C programmeerimiskeel dünaamiline, mistõttu võib sellega kohanemine aega võtta. Pealegi, Pythonis on suur hulk eeldefineeritud andmestruktuure ning nende käsitlemiseks mõeldud funktsioone. Pythoni andmestruktuuri ja -tüübi haldus on kergem, kui C programmeerimiskeeles. Samuti oleks mugavam erinevate hägurite implementeerimine teha Pythonis. Selleks, et server saaks UML-s ketta pealt loetud andmed kätte, konstrueeriti UML-s nii nimetatud klient, mis saadab serverile andmed ning saab selle peale vastuse. Peale hägurist läbilastud andmete kättesaamist jätkab UML oma tavapärast tööd ning saab jälgida, kas muudetud andmed põhjustavad ebareeglipärast tegevust või mitte.



Joonis 3.1. *Raamistiku andmevahetus.*

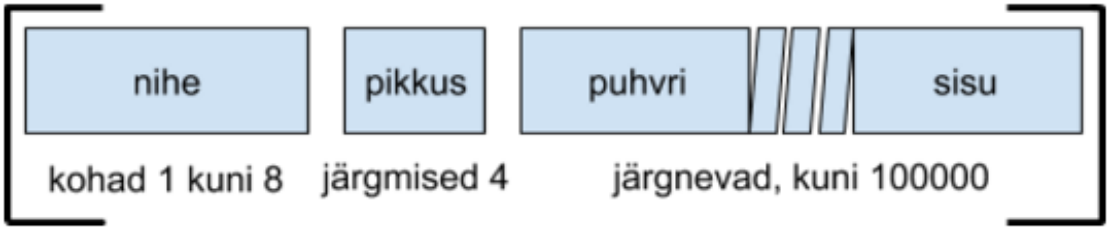
3.3 Raamistiku arendus

Arenduse läbiviimiseks paigaldati sülearvuti peale operatsioonisüsteem Linux Mint 18 "Sarah" – Cinnamon (64-bit), mis on vabavarana kättesaadav kodulehelt `https://linuxmint.com/edition.php?id=217`. Raamistiku arendamise käigus toimis antud arvuti hostina ehk virtuaalarvutit kandva arvutina. Host masinale seati üles User-mode Linux (vt jaotis 2.8), kuna see võimaldas vältida parandamatuid vigu ja sooritada kiiret plokkseadme emuleerimist, mis omakorda kiirendas katsete läbiviimist. Samuti paigaldati host masinale Python versiooniga 3.5, kuna Linuxi sisse ehitatud versioonist 2.7 ei piisanud.

3.3.1 Server

Serveri jaoks kirjutati Pythoni skript, mille käivitamise järel loodi host masinas datagrammsokkel (vt. jaotis 2.4), mis jäi ootama sissetulevaid andmeid. Iga ketta lugemise peale saadab kliendi pool serverile nihet, puhvri pikust ning puhvri väärtust, mis on ühendatud ühese järjestusega baidimassiivi. See jada on alati ühesugune. Baidijada elementide piltlik kujundus on toodud joonisel 3.2, jada ülesehitus vastab järgmistele reeglitele:

- esimesed 8 baiti kirjeldavad nihet;
- järgmised 4 baiti kirjeldavad puhvri pikkust;
- kõik järgnevad baidid kirjeldavat puhvri sees olevaid andmeid;
- terve baidijada pikkus on ülimalt 100000;
- kui puhver on pikem, siis võetakse puhvrist baite nii palju, kui mahub.



Joonis 3.2. *Kliendi käest saadav baidijada.*

Saadetava baidijada pikkus on seatud 100000 baidile ehk 100 kB. See on tehtud sellel põhjusel, et just sellist suurust oli võimalik staatiliselt allokeerida ilma mäluhalldusprobleemideta ning katsetused näitasid, et sellisest puhvripikkusest piisas, kuna nii suuri sidusaid lugemispäringuid ei tehtud. Nihe ja puhvripikkus on täisarvud. Nende baidiarvu erinevus tuleneb sellest, et kliendi poolel on vastavate andmete C tüübid (*C data type*) *unsigned long long*, pikkusega kaheksa baiti ja *unsigned int*, pikkusega neli baiti. Pärast andmete vastuvõtmist viib programm need vajalikule kujule. Vastavalt ülaltoodud reeglitele tükeldatakse baidijada kolmeks osaks ning salvestatakse muutujatesse järgnevate tegevuste lihtsustamiseks. Nihe ja puhvri pikkuse baidid teisendatakse *int*'ks ehk täisarvuks, kasutades *little-endian*'it (vt. jaotis 2.5), mis on protsessori arhitektuuri baidijärjestus. Puhver jääb jada tüüpi, kuna selle järjestust tuleb hägutestimise korral muuta. Programm jätab varem vastuvõetud nihked meelde, selleks, et oleks võimalik eristada korduvate mäluaadressite lugemist. Serveri poolel on võimalik rakendada erinevaid hägureid puhvri sisule ning saata vastus kliendile. Protsessi lihtsustamiseks on loodud abimeetod, mis konstrueerib vajaliku jadastruktuuriga vastuse, mille server saadab kliendile tagasi. Meetod nõuab kolme argumenti, millest kaks on *int* tüüpi ning kolmas on baidijada. Vastuse struktuur on järgmine:

- esimesed 8 baiti on vastuse kood, mis määrab ära kliendi edasised tegevused, hetkel on defineeritud kaks koodi (vt. tabel 3.1);
- järgnevad 8 baiti on esialgse puhvri pikkus;
- vastavalt vastuse koodile võib vastus sisaldada muudetud puhvrit

Tabel 3.1. *Vastuse koodid.*

Kood	Kliendi tegevus
0	Saadetav vastus ei sisalda puhvrit ning on 16 baiti pikk. Klient tegutseb edasi esialgse puhvriga, mingeid muudatusi ei toimu.
1	Saadetav vastus sisaldab, vastuse koodi, puhvri pikkust ning tühja puhvrit, mille pikkus võrdub kliendi poolt saadud puhvri pikkusega. Klient hakkab sooritama edasisi tegevusi saadud tühja puhvriga.

Iga õnnestunud andmete vastuvõtmise ja vastuse saatmise kohta annab server konsooli abil infot. Trükitakse välja saadud nihe, puhvri pikkus, kinnitus vastuse saatmise kohta ning saadetud vastuste arv. Puhvri sisu ei näidata, kuna siis muutuks konsooli väljund loetamatuks. Serveri käivitamiseks on vaja vastava kausta konsooliaknas sisestada käsk *python server.py*.

3.3.2 Klient UML-s

Kliendi osa on implementeeritud UML-s ühe funktsioonina, mida kutsutakse välja iga testitava ketta lugemise korral. Antud funktsioon võtab vastu kolm argumenti: puhvri viida, puhvri pikkuse ja nihke. Kõigepealt saadakse kätte serveri poolt tekitatud datagrammsokli fail, seotakse see kliendi aadressiga ning ühendatakse serveriga. Juhul kui mõni eelnevatest tegevustest ei õnnestunud, siis antakse sellest veasõnumiga konsooliaknas teada, suletakse soklifail kliendi poolel ning väljutakse funktsioonist. Sellisel juhul andmeedastust ei toimu, kuid selle protsessiga alustatakse uuesti järgmisel kettalugemisel. Kui ühendumine toimus veatult, siis ühendatakse argumendid ühtseks baidimassiiviks ning saadetakse serverile. Kusjuures, ühendamise ajal jälgitakse jaotises 3.3.1 selgitatud andmejada koostamise reegleid (vt joonis 3.2). Peale vastuse saamist kontrollitakse saadud jada pikkust ning kui see on vähemalt 16 elementi pikk, siis saadakse vastuse struktuuri reeglite alusel vastuse kood ning vastavalt sellele (vt. tabel 3.1) sooritatakse edasised tegevused. Kõige lõpus sulgeb klient soklifaili ning seejärel jätkub UML-i tavapärane töö.

3.4 Raamistiku kasutus

Sisuliselt on raamistikus implementeeritud mehhanism, mis teostab kettalugemise ajal automaatset andmevahetust UML-i ja Pythonit kasutava programmi vahel ning võimaldab mugavalt implementeerida erinevaid hägureid serveri poolel. Kogu arendus oli tehtud UML-i peal, seega kindlaim raamistiku kasutusviis oleks sama platvormi peal.

Eeldatav raamistiku kasutus näeks välja järgnevalt:

1. laetakse alla UML;

(a) laetakse alla kernel sobiva versiooniga (antud töös kasutati kerneli versiooni 4.11);

(b) pakitakse allalaetud kernel lahti;

2. UML-s asendatakse selle arenduse käigus muudetud failid:

   (a) kaustast `arch/um/`

      i. `drivers/ubd_kern.c`

      ii. `include/shared/os.h`

      iii. `os-Linux/file.c`

   (b) kaustast `server/`

      i. `server.py`

   (c) katse läbiviimisel, UML-i seadistamiseks sisestati käsud

   ```
   make defconfig ARCH=um
   make menuconfig ARCH=um
   ```

   kuid taaskordsel seadistamisel piisab faili `.config` lisamisest UML-i kausta

3. implementeeritakse vajalik hägur raamistiku serveri poolel;

4. käivitatakse raamistiku serveri pool, kaustas `/server` käsuga:

   ```
   python3.5 server.py
   ```

5. tekitatakse ketta pilt ning vormindatakse seda vajaliku Linuxi failisüsteemiga;

6. käivitatakse UML (enne esmakordset käivitamiset tuleb anda käsk UML-i kompileerimiseks)

   ```
   make ARCH=um
   ```

Korduvaks testimiseks, kui on tarvis teha uus hägur, siis piisab jätkamisest 3. punktist. Kui korduva testimise korral hägur jääb samaks, aga testitav failisüsteem muutub, siis on tarvis jätkata 5. punktist. Samasuguse häguri ja failisüsteemi korral testi kordamiseks piisab kui tegutsetakse 6. punktist.

4 Arutelu

4.1 Eesmärk

Antud peatükis võrreldakse selle töö raames saadud tulemust sarnaste lahendustega. Samuti räägitakse arenduse käigus esinenud suurematest raskustest ning arutatakse edasise töö võimalustest.

4.2 Olemasolevad lahendused

On olemas mitmeid hägutestimise vahendeid. Nad kõik erinevad oma mahu ning testitava objekti poolest. Näiteks erialaspetsiifiline *fuzzer* Microsoft SDL Regex Fuzzer on mõeldud regulaaravaldiste testimiseks vältimaks ReDoS-e (*Regular expression Denial of Service*). Samuti on olemas tasulised hägurid nagu Peach Fuzzing Platform. Need on keerulise loogikaga ehitatud fuzzerid, milledel on tavaliselt olemas oma API (*Application programming interface*). Peach Fuzzing Platformi korral on tegu rakendusega, mida kasutavad sellised ettevõted nagu Boeng, Intel, Oracle, ning isegi United States Department of Homeland Security. Selle töö eesmärki, failisüsteemi *fuzzimist* silmas pidades, on kõige sarnasem lahendus hägur nimega AFL või American Fuzzy Lop. See on muteeriv hägur, mis kasutab väga palju jagatud mälu, kuid selle eest on ta väga tõhus. AFL-ga on tehtud ka failisüsteemi hägutestimist, millest rääkisid Vegard Nossum ja Quentin Casasnovas oma esitluses Vault2016 konverentsil [NC16].

Peamine erinevus antud töö tulemuse ja AFL-i vahel on see, et AFL kasutaks erinevaid virtuaalkettaid, kus on juba häguandmed peale kirjutatud. Iga testseansi jooksul serveeriks AFL sama ploki kohta alati ühte ja sama failisüsteemi kujutist. Käesoleva raamistiku abil oleks aga võimalik mitmekordse ühe ja sama ploki lugemise kohta serveerida erinevat sisu. Tuleb tähele panna seda, et selle töö raames konkreetset hägurit loodud ei ole, vaid ehitati karkass, mis on mõeldud failisüsteemide hägutestimise lihtsustamiseks.

4.3 Arenduse käigus tekkinud raskused

Peamine raskus esines andmete saatmises raamistiku UML-ist hägurile. Esimese lahendusena üritati manustada Pythonis kirjutatud funktsioonid C koodi sisse. Selleks tuli lisada teatud UML-i faili päisesse *#include Python.h*. Host masinas olevas lihtsas näidisprogrammis õnnestus seda teostada, kuid sama koodi lisamisel UML-i, andis see kompileerimise ajal veateadet, mis ütles, et vastavat faili ei leidu. Host masina Python.h faili lisamine ei lahendanud probleemi, kuna Python.h failis esinesid omakorda viited teistele failidele, mis puudusid UML-i kataloogis ning kõigi nende failide läbitöötamine osutus ebarealistlikuks. Teiseks lahenduseks oli implementeerida andmevoo sokkel (*stream socket*), mis oleks kindlam lahendus, kui datagrammsokkel, kuna andmevoo

sokli korral toimetatakse andmed kohale nende saatmise järjekorras ning ebaõnnestunud andmeedastuse korral katkestatakse ühendus automaatselt. See lahendus töötas samuti lihtsa programmi korral ning UML-i kompileerimine toimus vigadeta, kuid selle käivitamine ei õnnestunud, sest protsess jäi kinni ning selle töö tuli käsitsi lõpetada. Tõrkeanalüüsi käigus avastati, et UML kasutas *-nostdinc* lippu, mis tähendab *no standard include*, mille tõttu UML ei kasutanud standardseid teeke. Lõpuks lahendati probleem datagrammsokliga, mis võimaldas käsitsi ära määrata serveri ja kliendi aadressid ning selle, kuidas ja millal ühendust teostada.

Teine raskus, mis ei saanud lahendust oli see, et kui sisestati UML-s käsk ketta pealt andmete lugemiseks, siis klient ei saatnud neid andmed serverile. Arvatavasti viga seisneb selles, et kliendi käivitamiseks mõeldud funktsiooni ei kutsuta vajalikus kohas välja. Seetõttu saab Linuxi failisüsteeme testida ainult UML-i käivitamise ajal.

4.4 Edasine töö

Raamistiku põhiline osa, server, oli kirjutatud Pythoni abil, mis on võrreldes C-ga suhteliselt kergelt õpitav programmeerimiskeel. See tähendab, et erinevate hägurite või muu funktsionaalsuse lisamine raamistikku on selle võrra lihtsam. Käesolev töö saab olla aluseks teistele lõputöödele või iseseisvatele uurimustele, kus sooritatakse mingi konkreetse failisüsteemi testimist ning uuritakse tulemusi. Ühtlasi on võimalik ka raamistiku funktsionaalsust laiendada, lisades serveri poole tõrgete logimist või ehitada raamistiku sisse hägurit, mida hakatakse vaikimisi kasutama. Lisaks jääb edasise töö jaoks eelmises punktis kirjeldatud teise raskuse lahendamine.

Kokkuvõte

Käesoleva töö raames ehitati raamistik, mis võimaldab hägutestimise abil otsida erinevatest Linuxi failisüsteemidest vaheletrügimisnõrkusi. Raamistik koosneb kahest osast. Esimene osa oli konstrueeritud User-mode Linuxis ning see on mõeldud iga ketta lugemise peale saatma baidijadu hägurile. Teine, serveri osa oli kirjutatud Pythoni abil. Selle abil on võimalik implementeerida erinevaid hägureid, millega saaks User-mode Linuxist saadud baidijadu teisendada. Samuti on olemas funktsionaalsus, millega saab teisendatud andmeid UML-le tagasi saata ning kus saab ära määrata UML-i edasised tegevuse nende andmetega. Raamistiku tööpõhimõtet kontrolliti katsega, kus õnnestus käigupealt otsustada, milliste sektorite sisu vahetada nullide vastu ning sööta UML-s jooksvale Linuxile.

Arenduse käigus keskenduti Linuxi failisüsteemile Ext4, aga ramistiku töökindlus ei sõltu testitavast failisüsteemist, kuna failisüsteemi määrab ära testimises kasutatav ketas. Seega, teoreetiliselt peab raamistik töötama suvalise Linuxi failisüsteemiga. Antud töö tulemusena saab luua häguri, mis võimaldaks Linuxi failisüsteemidest leida TOCTOU tüüpi turvaauke. Samuti on pandud alus edasisele arendustööle, mille käigus oleks võimalik laiendada raamistiku funktsionaalsusi.

Viidatud kirjandus

[Ass04] USENIX Association. Proceedings of the 13th usenix security symposium. https://www.usenix.org/legacy/events/sec04/tech/full_papers/dean/dean.pdf, 2004. kasutatud 09.05.2017.

[BBB10] Christoph Baumann, Bernhard Beckert Holger Blasum, and Thorsten Bormer. Ingredients of operating system correctness* lessons learned in the formal verification of pikeos. Technical report, Saarland University, Dept. of Computer Science and Karlsruhe Institute of Technology and SYSGO AG, 2010.

[Dik] Jeff Dike. User-mode Linux homepage, http://user-mode-linux.sourceforge.net/ kasutatud 09.05.2017.

[Fou16] FreeŠoftware Foundation. *GNU Make Manual*. Free Software Foundation, Inc., 2016. https://www.gnu.org/software/make/manual/ kasutatud 09.05.2017.

[Gol10] Golftheman. File:operating system placement.svg. https://commons.wikimedia.org/wiki/File:Operating_system_placement.svg, 2010. kasutatud 09.05.2017.

[Gui10] Dan Guido. *Fuzzing*, 2010. https://fuzzinginfo.files.wordpress.com/2012/05/fuzzingintro_fall2010.pdf kasutatud 09.05.2017.

[Lee15] Thorsten Leemhuis. Linux-kernel durchbricht die 20-millionen-zeilenmarke. *Heise Online*, 2015.

[Lev15] Sasha Levin. Filesystem fuzzing. At the Vault2015 conference, 2015. https://lwn.net/Articles/637151/ kasutatud 09.05.2017.

[McC04] Steve McConnell. *Code Complete*, chapter How Many Errors Should You Expect to Find?, page 558. Microsoft Press A Division of Microsoft Corporation One Microsoft Way Redmond, Washington 98052-6399, second edition, 2004.

[MIT] MITRE. Time-of-check time-of-use (toctou) race condition. https://cwe.mitre.org/data/definitions/367.html. kasutatud 09.05.2017.

[MIT17] MITRE. Linux kernel vulnerability statistics. http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html, 2017. kasutatud 09.05.2017.

[NC16] Vegard Nossum and Quentin Casasnovas. Filesystem fuzzing with american fuzzy lop. At Vault2016 conference, 2016.

[Nem16] Andrej Nemec. Multiple race conditions in the ext4 filesystem implementation in the linux kernel before 4.5. Red Hat bugreport, apr 2016. https://bugzilla.redhat.com/show_bug.cgi?id=1323577 kasutatud 09.05.2017.

[oIS13] UNCŠchool of Information and Library Science. Hard drive structure. https://ils.unc.edu/courses/2013_spring/inls525_001/slides/wk03.html, 2013. kasutatud 09.05.2017.

[PHR+09] Donald E. Porter, OwenŠ. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. Technical report, Department of Computer Sciences, The University of Texas at Austin, 2009. http://www.sigops.org/sosp/sosp09/papers/porter-sosp09.pdf kasutatud 09.05.2017.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing Brute Force Vulnerability Discovery*. Pearson Education, Inc., 2007.

[SGC+09] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. Technical report, Stony Brook University and IBM T.J. Watson Research Center, 2009. http://www.fsl.cs.sunysb.edu/docs/valor/valor_fast2009.pdf kasutatud 09.05.2017.

[Sha07a] R.Š. Shaw. File:big-endian.svg. https://commons.wikimedia.org/wiki/File:Big-Endian.svg, 2007. kasutatud 09.05.2017.

[Sha07b] R.Š. Shaw. File:little-endian.svg. https://commons.wikimedia.org/wiki/File:Little-Endian.svg, 2007. kasutatud 09.05.2017.

[UMD] UMD Department of Computer Science. *Big and Little Endian*. https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html kasutatud 09.05.2017.

[wik17a] Kernel wiki. *Ext4 Disk Layout*, 2017. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout kasutatud 09.05.2017.

[wik17b] Kernel wiki. *Ext4 Howto*, 2017. https://ext4.wiki.kernel.org kasutatud 09.05.2017.

[Wik17c] Wikipedia. List of file systems. https://en.wikipedia.org/wiki/List_of_file_systems, 2017. kasutatud 09.05.2017.

[WP05] Jinpeng Wei and Calton Pu. Tocttou vulnerabilities in unix-style file systems: An anatomical study. https://www.usenix.org/legacy/event/fast05/tech/full_papers/wei/wei.pdf, 2005. kasutatud 09.05.2017.

Lisa A: Skriptid

Arenduse käigus loodud kood asub Githubi repositooriumis aadressil `https://github.com/VladAlenitsev/fuzzing`.

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, **Vladislav Alenitsev**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose

   Failisüsteemi hägutestimise raamistik

   mille juhendajad on Meelis Roos ja Kristjan Krips

   1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

   1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 10.05.2017