TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Paul Jürgens IAIB223038

Spetsialiseeritud C++ andmevootöötluse teegi kasutamine Intel SGX-i ja Rusti iteraatorite kasutamine Intel TDX-i keskkonnas: võrdlev analüüs

Bakalaureusetöö

Juhendaja: Gert Kanter
PhD

Kaasjuhendaja: Armin Daniel Kisand
MSc

Tallinn 2025

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Paul Jürgens

12.05.2025

Annotatsioon

Lõputöö eesmärk on võrrelda spetsialiseeritud C++ andmevootöötluse teegi kasutamist Intel SGX-i keskkonnas ja Rusti ning Rusti iteraatorite kasutamist Intel TDX-i keskkonnas. Töö keskendub nende kahe platvormi erinevustele jõudluses ning arenduskogemuses sama andmetöötlusülesande implementeerimise korral.

Kuna Intel TDX pakub teoorias mitmeid eeliseid Intel SGX-i ees, aga praktikas pole selge, kas see oleks sobiv alteratiiv Intel SGX-i lahendusele, siis on vajalik viia läbi praktiline eksperiment nende kahe lähenemise võrdlemiseks.

Eksperimendi käigus teostati mõlemas keskkonnas identne andmetöötlusvoog, mis hõlmas kettalt andmete lugemist ja kirjutamist, krüptograafilisi operatsioone ning andmete muutmist. Eksperimentaalsed mõõtmised tõid esile mõlema lähenemisviisi tugevused ja nõrkused, pakkudes praktilisi soovitusi sobiva tehnoloogia valimiseks.

Kuigi Intel TDX-i lahendus pakub kaasaegsmat arenduskogemust ja paremat jõudlust, osutus selle seadistamine keerukaks peamiselt ebaküpse ja pidevalt areneva tarkvaralise ökosüsteemi tõttu. Seevastu Intel SGX-i lahendus pakub juba optimeeritud raamistikku turvatud rakenduste loomiseks, kuid kannatab piiratud dokumentatsiooni ja tülikama arenduskogemuse tõttu.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 33 leheküljel, 6 peatükki, 16 joonist, 4 tabelit.

Abstract

Comparing a Specialized C++ Stream Processing Library for Intel SGX with Rust Iterators on Intel TDX

The objective of this thesis is to compare the use of a specialized C++ stream processing library within the Intel SGX (Software Guard Extensions) environment to the use of Rust and Rust iterators in the Intel TDX (Trust Domain Extensions) environment. The focus lies in evaluating the performance and developer experience of both approaches.

Intel TDX is often considered a more advanced and user-friendly successor to Intel SGX, offering several improvements. However, its practical suitability as an alternative to the Intel SGX based solution discussed in this thesis remains unclear. This thesis seeks to clarify this by implementing and benchmarking equivalent data analysis pipelines using both technologies for a direct comparison.

The data processing pipeline implementation involves reading from and writing to disk, performing cryptographic operations, and transforming data in memory. Both implementations are designed to be functionally equivalent to ensure a fair comparison. Measurements were taken to evaluate the time required to perform these operations, along with qualitative factors such as ease of implementation, amount of boilerplate code, learning curve, and compiler error clarity.

The results highlight the strengths and weaknesses of both solutions. While the Intel TDX solution offers a modern programming experience and improved performance, setting it up proved challenging primarily due to its immature and evolving software ecosystem. In contrast, the Intel SGX solution provides an already optimized framework for building data processing tasks, but suffers from a convoluted ecosystem, limited documentation, and a more cumbersome developer experience.

Based on the experimental findings, this thesis offers practical guidance for selecting the appropriate technology when developing a secure data processing application.

The thesis is written in Estonian and is 33 pages long, including 6 chapters, 16 figures and 4 tables.

Lühendite ja mõistete sõnastik

ECDF	Empiiriline kumulatiivne jaotusfunktsioon (*Empirical Cumulative Distribution Function*)
GB	Gigabait (*Gigabyte*)
GCC	GNU kompilaatorite kollektsioon (*GNU Compiler Collection*)
HI	Riistvaraline isolatsioon (*Hardware Isolation*)
Intel SGX	Intel® Software Guard Extensions
Intel TDX	Intel® Trust Domain Extensions
IO	Sisend-väljund (*Input-Output*)
MNO	Mobiilisideoperaator (*Mobile Network Operator*)
OS	Operatsioonisüsteem (*Operating System*)
TEE	Usaldatav täitmiskeskkond (*Trusted Execution Environment*)
VM	Virtuaalmasin (*Virtual Machine*)

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Turvaliste andmetöötluslahenduste kasutamine on kriitilise tähtsusega valdkondades, kus töödeldakse tundlikku teavet, näiteks finantssektoris, tervishoius ja sageli ka pilveteenustes. Intel SGX (Software Guard Extensions) [1] ja Intel TDX (Trust Domain Extensions) [2] on tehnoloogiad, mis võimaldavad luua turvalisi keskkondi, kus andmete ja koodi konfidentsiaalsust saab kaitsta ka siis, kui ülejäänud süsteem on ohustatud. Need tehnoloogiad on mõeldud lahendama probleemset olukorda, kus kolmandad osapooled, näiteks pilveteenuste pakkujad, võivad omada juurdepääsu süsteemi tasandile, kuid andmetöötluse konfidentsiaalsus peab olema säilitatud.

Ettevõte Cybernetica on loonud Intel SGX-i turvatehnoloogial põhineva toote Sharemind HI [3], mis kasutab spetsiaalselt selleks lahenduseks loodud andmevootöötluse teeki Streams [4]. Toode võimaldab Streams teegi abil luua teenuseid, mis jooksevad Intel SGX-iga turvatud keskkonnas.

Kuid Sharemind HI kasutamisega kaasnevad mitmed arendusprotsessi raskendavad tegurid, mis tulenevad Intel SGX-i, C++-i ning Streams teegi eripäradest. Neid arenduskogemust mõjutavaid aspekte on käsitletud lähemalt peatükkides 2.1.1 ja 5.

Potentsiaalse alternatiivina saaks kasutada lahendust, mis põhineb Intel TDX-il koos Rusti ja Rusti iteraatoritega. Selline kombinatsioon võiks pakkuda arendajale sujuvamat ja kaasaegsemat kogemust turvaliste programmide arendamisel. Lisaks on Rusti iteraatorid laialdaselt kasutusel ja paremini dokumenteeritud kui ettevõttesisene Streams teek, mis võib veelgi lihtsustada arendustööd.

Probleem seisneb selles, et praktikas on Intel TDX-i lahenduse kasutatavus küsitav, kuna mitmed võtmeküsimused jäävad ebaselgeks. Pole teada, milline on Intel TDX-i lahenduse jõudlus ning arenduskogemus võrreldes Intel SGX-i põhise lahendusega elulises andmetöötlusstsenaariumis. Näiteks võib Rusti kasutamine pakkuda mugavamat mälu- ja veakäsitlust, kuid kui see tuleb liigselt programmi arvutuskiiruse arvelt, võib migratsioon osutuda kahjulikumaks kui algselt arvatud.

Lõputöö eesmärk on täita need lüngad ning pakkuda Cyberneticale teavet, mis aitab teha teadlikke valikuid optimaalse platvormi kasutuse osas tulevikus. Täpsemalt soovitakse leida vastus küsimustele:

Uurimisküsimus: Kas Intel TDX-i lahendus, mis kasutab Rusti ja Rusti iteraatoreid on elulises andmetöötlusolukorras sobiv alternatiiv Intel SGX-i lahendusele, mis kasutab C++-i, Sharemind HI-d ja Streams teeki?

Alaküsimus 1: Kas andmetöötluse teostamiseks vajalik funktsionaalsus on olemas?

Alaküsimus 2: Kuidas erineb jõudlus?

Alaküsimus 3: Kuidas erineb arenduskogemus kasutatavuse ja laiendatavuse seisukohalt?

Nendele küsimustele vastamiseks viiakse läbi praktiline eksperiment, mis võrdleb mõlema lahenduse jõudlust ja arenduskogemust. Autor loob kummagi lahenduse jaoks jõudlustestid, millega mõõdetakse ajakulu erinevate andmete töötlemisel. Seejärel võrreldakse ja analüüsitakse mõõtmistulemusi. Samuti koostatakse põhjalik analüüs erinevustest arenduskogemuses.

Töö alguses tutvustatakse tausta, puudutades teemasid, nagu usaldatavad täitmiskeskkonnad ning kasutatud tööriistad ja tehnoloogiad. Seejärel kirjeldatakse jõudlustestide disainimetooditkat ning implementatsiooni mõlema lahenduse puhul. Peale seda näidatakse ja analüüsitakse jõudlustestide tulemusi ning lõpetuseks võrreldakse erinevusi arenduskogemuses.

2 Taust

Selles peatükis on selgitatud vajalikud taustateadmised, mis on aluseks lõputöö praktilise osa teostusele.

2.1 Usaldatav täitmiskeskkond

Usaldatav täitmiskeskkond (ingl *Trusted Execution Environment*, TEE) on spetsiaalne protsessori käsustiku laiendus, mille eesmärk on pakkuda täiendavat turvalisust tundlike andmete töötlemiseks ja koodi täitmiseks [5]. TEE on usaldusväärne keskkond, kus pahatahtlikud programmid ning isegi operatsioonisüsteem ei saa mõjutada selle sees toimuvaid tegevusi ega mälus kasutusel olevaid andmebaite lugeda. Seda saavutatakse tavaliselt riistvaralise isolatsiooni ja spetsiifiliste turvamehhanismide abil.

Sageli on TEE peamine eesmärk kaitsta kriitilisi andmeid, nagu krüptovõtmed, autentimisinfo või maksetehingute andmed, mille lekkimine võib kaasa tuua tõsiseid tagajärgi. Näiteks mobiiltelefonides kasutatakse TEE-sid kõrgete turvanõuetega rakenduste (nt Google Pay või Apple Pay) töökindluse tagamiseks [6], [7].

TEE põhiomadused:

- **Turvaline täitmine ja isoleeritus**: Võimaldab koodi turvalist täitmist isoleeritud keskkonnas, kaitstes volitamata juurdepääsu või manipuleerimise eest operatsioonisüsteemi, süsteemi administraatori või teiste rakenduste poolt.
- **Konfidentsiaalsus**: Tagab, et tundlikud andmed jäävad krüpteerituks ja volitamata osapooltele kättesaamatuks.
- **Atesteerimine**: Pakub krüptograafilist tõendit, et TEE käitab usaldusväärset ja muutmata tarkvara, võimaldades kaugosapooltel kontrollida TEE terviklikkust.

Populaarsed TEE tehnoloogiad on näiteks ARM TrustZone, Intel SGX ja Apple Secure Enclave, mis on laialdaselt kasutusel näiteks mobiilseadmetes ja pilveteenustes. Need tehnoloogiad pakuvad turvalist keskkonda tundlike andmete ja kriitiliste rakenduste kaitseks, aidates vähendada rünnakuvõimalusi ning suurendada süsteemide usaldusväärsust [8].

Mitmed tuntud pilveteenustepakkujad näiteks Google ja Microsoft töötavad aktiivselt, et integreerida TEE tehnoloogiale toetuvaid konfidentsiaalseid andmetöötluslahendusi oma pakutud teenuste hulka [9], [10].

Selle bakalaureusetöö raames käsitletakse lähemalt vaid Intel SGX ja Intel TDX TEE tehnoloogiaid.

2.1.1 Intel Software Guard Extensions

Intel SGX (Software Guard Extensions) on riistvarapõhine TEE tehnoloogia, mis võimaldab luua nii-öelda tarkvaralisi enklaave (ingl *enclave*) [1]. Enklaav hõlmab vaid ühte programmimoodulit (*.so* faili) ning jagab programmi usaldatud ja mitteusaldatud osadeks. Usaldatud osa, mis jookseb enklaavi sees, vajab sisend-väljund (ingl *Input-Output*, IO) operatsioonide jaoks mitteusaldatud osa. Näiteks andmete saatmiseks TEE keskkonda peab mitteusaldatud programmiosa need edastama usaldatud osasse.

Intel SGX-i üheks peamiseks eeliseks teiste TEE tehnoloogiate ees on selle väike ründepind (ingl *attack surface*) ning minimaalne usaldatav töötlusbaas (ingl *Trusted Computing Base*, TCB). Lisaks toetab Intel SGX kaugatesteerimist, kasutades selleks Intel SGX DCAP teeki, mille abil saavad kaugkasutajad kontrollida enklaavi terviklikkust ja autentsust [11]. Samuti võimaldab Intel SGX andmete turvalist salvestamist kettale, tagades, et salvestatud andmetele pääseb ligi vaid sama enklaavi kaudu [12]. Tänu nendele omadustele sobib Intel SGX mitmesugusteks kasutusjuhtudeks, sealhulgas pilvepõhiseks andmetöötluseks ja privaatsust säilitavateks arvutusteks.

Kuid Intel SGX-il on ka puudusi. Esiteks mitme lõime kasutamine Intel SGX keskkonnas võib potentsiaalselt muuta enklaavi haavatavaks rünnakute suhtes, mis tähendab, et keerukamate rakenduste korral võib Intel SGX-i turvamehhanism olla puudulik [13]. Lisaks on Intel SGX-i arendamine keerukas, nõudes spetsiifilist keskkonda ja eriteadmisi. Olemasoleva rakenduse Intel SGX-i enklaavi juurutamine (ingl *deploy*) vajab koodi ümber kirjutamist, kuna programm peab jaotuma eelnevalt mainitud viisil usaldatud ja mitteusaldatud osadeks. Selle probleemi leevendamiseks on mitmeid lahendusi, näiteks Gramine projekt, mille kasutamisel peab koodi märgatavalt vähem (kui üldse) muutma [14].

Kuigi Intel SGX pakub tugevat turvalisust, on aja jooksul leitud ka riistvarapõhiseid turvanõrkusi, nagu Spectre ja Meltdown [13], [15].

2.1.2 Intel Trust Domain Extensions

Intel TDX (Trust Domain Extensions) on Inteli uusim riistvarapõhine konfidentsiaalse andmetöötluse lahendus [2]. Tegemist on TEE tehnoloogiaga, mis võimaldab luua isoleeritud ja turvatud virtuaalmasinaid (ingl *Virtual Machine*, VM), mida nimetatakse usaldusdomeenideks (ingl *Trust Domain*, TD).

Intel TDX pakub sarnast funktsionaalsust ning turvalisust, mis Intel SGX. Lisaks lahendab

see mitmed Intel SGX-i puhul esile tulnud puudus [16]. Näiteks kuna Intel TDX-i puhul puudub vajadus jaotada programm usaldatud ja mitteusaldatud osadeks, on võimalik rakendusi juurutada TEE-sse, ilma et peaks tegema koodile muudatusi. See lihtsustab märgatavalt Intel TDX-i kasutuselevõttu olemasolevate rakenduste puhul.

Lisaks võimaldab Intel TDX turvalisemalt käitada mitmel lõimel töötavaid protsesse võrreldes Intel SGX-iga. Selle põhjuseks on asjaolu, et Intel SGX-i puhul saab ründaja mõjutada lõimede plaanuri (ingl *thread scheduler*) tööd, mida on võimalik ära kasutada erinevate rünnakute läbiviimiseks [17]. Intel TDX-i puhul asub lõimede plaanur aga usaldatud virtuaalmasina sees, mistõttu on see kaitstud väliste mõjutuste eest.

2.2 Sharemind HI

Sharemind HI (*Hardware Isolation*) on Cybernetca AS poolt arendatud C++-ile suunatud privaatsust säilitav andmetöötlusplatvorm, mis kasutab andmete turvaliseks käsitluseks Intel SGX-i [3]. Sharemind HI lihtsustatud töövoog:

1. Sharemind HI olukorraspetsiifiline lahendus seatakse serveris üles.
2. Andmeomanikud teostavad kaugatesteerimise Sharemind HI serveriga.
3. Andmeomanikud krüpteerivad oma andmed ning saadavad need Sharemind HI serverisse.
4. Sharemind HI server saab andmed kätte, kusjuures väljaspool Intel SGX enklaavi pole andmeid võimalik lahti krüpteerida ning krüpteeritud andmetest ei ole võimalik midagi järeldada algsete andmete kohta.
5. Sharemind HI server töötleb andmed Intel SGX TEE-s.
6. Sharemind HI server tagastab töötluse tulemuse krüpteeritud kujul.

Sharemind HI on sisuliselt raamistik, mis lihtsustab Intel SGX-i jaoks mõeldud andmetöötlusprogrammide loomist, abstraheerides suuresti Intel SGX-iga kaasnevat lisakeerukust arendusprotsessis. Spetsiifiliselt võimaldab Sharemind HI andmeanalüüsi olukorras, kus andmeomanikke on mitu ning iga osapoole andmed peavad jääma salastatuks. Lahenduse oluliseks osaks on ka Intel SGX-i atesteerimisfunktsionaalsus, mis võimaldab klientidel kinnitada, et server töötab õigete rakenduste, konfiguratsioonide ja tarkvaraversioonidega.

2.2.1 Streams teek

Streams on Cybernetica AS poolt arendatud C++ teek, mis kaasneb Sharemind HI lahendusega [4]. Streams teek on mõeldud efektiivseks voopõhiseks andmetöötluseks piiratud mäluga keskkondades, näiteks Intel SGX enklaav. See teek võimaldab andmetöötlust ilma, et peaks kõik andmed korraga enklaavi mällu lugema, pakkudes erinevaid abimeetodeid, mis lihtsustavad andmete käsitlemist ja muutmist.

Abimeetodid Streams teegis kuuluvad ühte kolmest kategooriast:

1. ***Sources***: Seda tüüpi meetodid genereerivad elemente. Näiteks andmete sisselugemist failist teostatakse *source* meetodi abil.
2. ***Sinks***: Seda tüüpi meetodid tarbivad sisendelemendid ning lõpetamisel kas tagastavad mingi üksiku väärtuse või ei tagasta üldse midagi (*void*). Näiteid *sink* meetoditest: `foreach` ja `collect`.
3. ***Pipes***: Seda tüüpi meetodid töötlevad sisendelemente, saades neid muuta, eemaldada või lisada. Nende meetodite tulemust saab edasi suunata *pipe* või *sink* meetodisse, mis võimaldab luua andmekonveieri (ingl *data pipeline*) laadse töötlusloogika. Näiteid *pipe* meetoditest: `filter`, `smap` ja `join`.

Streams teek pakub kokku 31 unikaalset abimeetodit andmetöötluse teostamiseks. Eri kategooriatest meetodeid saab omavahel põimida kasutades *pipe* operaatorit (»=). Joonisel 1 on näide lihtsast Sharemind HI andmetöötlusloogikast, mis kasutab Streams teeki kahe hulga ühisosa leidmiseks ning tulemusele suvalise teisendusfunktsiooni rakendamiseks.

```
void run(TaskInputs const & inputs, TaskOutputs & outputs) {
    enclave_printf_log("Running set intersection task!");

    stream::join(
        // This is how encrypted inputs from input providers are read
        // using the Streams API. Sharemind HI automatically decrypts
        // the input data.
        stream::into<int32_t>(inputs, "input1"),
        stream::into<int32_t>(inputs, "input2"),
        // These lambdas are used to extract joinable keys on more
        // complex structures.
        [](int32_t el){return el;},
        [](int32_t el){return el;}
    ) >>=
    stream::smap([](auto const & joined_pair) {
        // Some arbitrary mapping function.
        return joined_pair.first[0];
    }) >>=
    // Encrypt and persist the analysis result, so it can be accessed
    // (only) by the designated output consumers. Sharemind HI
    // transparently handles the key management in the background.
    stream::encryptedOutput(outputs, "output");
}
```

Joonis 1. Sharemind HI Streams teegi koodinäide.

2.3 Rusti iteraatorid

Rust on kaasaegne programmeerimiskeel, mis on loodud pakkuma head jõudlust, mäluturvalisust ja paralleelsust ilma koristita (ingl *garbade collector*) [18]. See on saanud populaarseks alternatiiviks C ja C++ kõrval, eriti tänu oma võimele ennetada mäluvigade

tekkimist jõudlust ohverdamata.

Rusti iteraatorid võimaldavad sarnaselt Streams teegile efektiivset andmetöötlust. Rusti iteraatorite aluseks on *next* meetod, millega saab andmejadast laisalt väärtusi ükshaaval töödelda [19], [20]. See on samuti kasulik andmetöötluseks piiratud mäluga keskkondades. Iteraatorid on Rustis implementeeritud sarnaselt voogudega teistes programmeerimiskeeltes, kuid on tihedalt seotud Rusti eripärase tüübi süsteemiga (ingl *type system*).

Rusti iteraatorite abimeetodeid saab jagada kaheks [21]:

1. **Adapterid (ingl *adaptors*)**: funktsioonid, mis võtavad sisendiks iteraatori ja tagastavad samuti iteraatori. Sarnane Streams teegi *pipe* meetoditele. Näiteks `map` ja `filter`.
2. **Tarbijad (ingl *consumers*)**: funktsioonid, mis võtavad sisendiks iteraatori ja tagastavad mingi lõpliku väärtuse. Sarnane Streams teegi *sink* meetoditele. Näiteks `collect` ja `fold`.

Sarnaselt Streams teegile pakuvad Rusti iteraatorid meetodite komplekti, mida saab omavahel kokku aheldada. Joonisel 2 on näide lihtsast iteraatorite kasutusest. Funktsionaalsuse poolest on see koodinäide sarnane Streams teegi näitega joonisel 1: leitakse kahe hulga ühisosa ning tulemusele rakendatakse suvaline teisendusfunktsioon.

```
fn run(input1: &HashSet<i32>, input2: &HashSet<i32>) -> Vec<i32> {
    println!("Running set intersection task!");

    input1
        .intersection(input2)
        .map(|&el| el)  // Some arbitrary mapping function
        .collect()
}
```

Joonis 2. Rusti iteraatorite koodinäide.

2.3.1 Itertools teek

Itertools on Rusti teek, mis laiendab standardteegi iteraatorite funktsionaalsust, lisades uusi meetodeid ja makrosid [22]. Lisatud meetodid on näiteks *chunk_by*, *sorted_by* ja *merge_join_by*.

Keerulisema andmetöötlusloogika implementeerimiseks võib Rusti standardteegi iteraatorite funktsionaalsus olla ebapiisav. Täpselt selleks juhuks ongi loodud Itertoolsi teek. Kui lisaks Rusti standardteegi iteraatoritele kasutada Itertoolsi, siis on arendajal võimalik kasutada rohkem kui 100 unikaalset abimeetodit andmetöötluse teostamiseks.

2.4 Eurostat-Cybernetica Sharemind HI projekt

Aastatel 2020 kuni 2021 viis Eurostat koostöös Cyberneticaga läbi projekti, kus uuriti, kuidas teostada turvalist ja privaatsust säilitavat mobiilsideoperaatorite (ingl *Mobile Network Operator*, MNO) andmete töötlust [23]. Selle projekti jaoks valitud lahendus põhines TEE tehnoloogial koos riistvaralise isolatsiooniga. Projekti ametlik pealkiri on: "A proof-of-concept solution for the secure private processing of longitudinal Mobile Network Operator data in support of official statistics (ESTAT 2019.0232)".

Projekti peamine eesmärk oli tõestada, et sellise suuremahulise andmetöötlusülesande lahenduseks on võimalik privaatsuse säilitamiseks kasutada TEE tehnoloogiaid [24]. Üldisemalt sooviti leida vastus järgnevatele küsimustele:

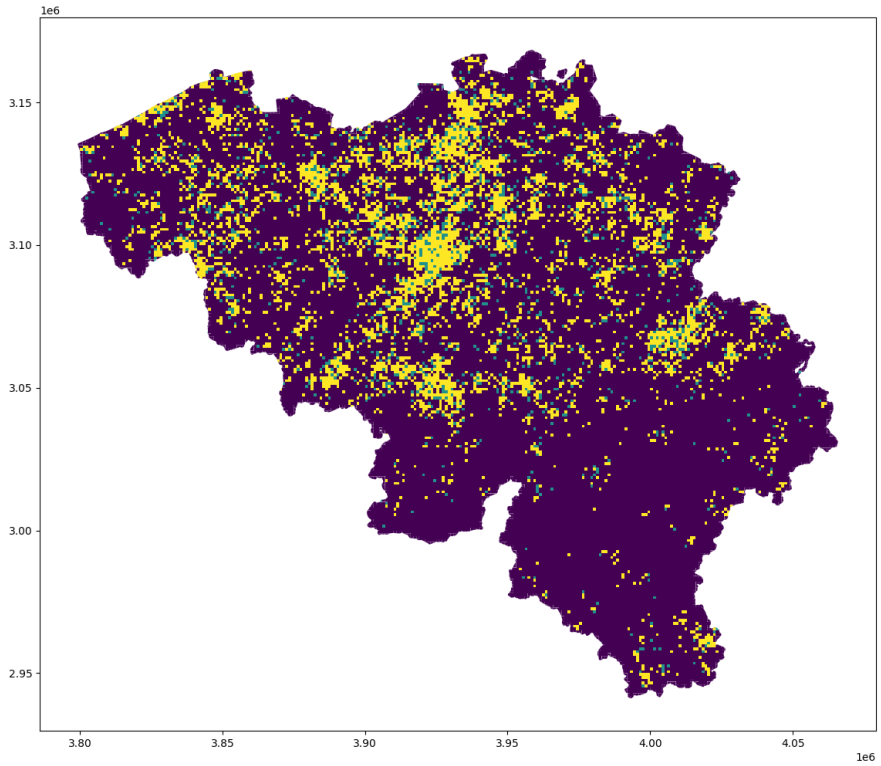
1. Kas TEE lahendused on piisavalt skaleeritavad, et tulla toime andmemahtude ja -kiirustega, mis esinevad elulises MNO andmetöötlusolukorras?
2. Kas TEE lahendused on piisavalt küpsed ja paindlikud, et neid saaks kasutada elulise andmetöötluse töövoo jaoks?
3. Milliseid juriidilisi aspekte hõlmab TEE lahenduste kasutuselevõtt?

Nende küsimuste adresseerimiseks formuleeris Eurostat hüpoteetilise elulise stsenaariumi, mida järgides arendas Cybernetica AS ka prototüübi [25]. Tulemuseks loodud projekt kasutas andmetöötluse turvalisuseks Sharemind HI-d (Intel SGX) koos Streams teegiga.

Prototüüp töötles MNO-lt saadud inimeste asukohaandmeid. Töötluse väljundiks on summeeritud ja korrastatud andmed inimeste ligikaudsete asukohtade kohta mingi ajaperioodi jooksul. Projekti raames loodud andmeid visualiseerivat soojakaarti (ingl *heatmap*) on näha joonisel 3. Soojemad toonid tähistavad piirkondi, kus on inimesed rohkem aega veetnud ning külmemad toonid vastupidist.

Prototüübis esineb järgnev andmetöötlusloogika (lugeja huvides lihtsustatud) [26]:

1. Rakendus saab sisendandmetena mobiilsideoperaatorilt (MNO) andmed nende teenuse kasutajate kohta. Edastatud andmed kirjeldavad kasutajate ligikaudseid asukohti kindlal kuupäeval.
2. Rakendus töötleb andmed (sorteerib, puhastab, parandab jne).
3. Tulemuse faili kirjutamine:
    a) Kui tegemist on esimese töötlusega, siis kirjutatakse tulemus faili.
    b) Kui tulemuste fail on juba olemas (st pole esimene töötlus), siis eelnev tulemus ühendatakse uue sisendiga (*outer join*) ning seejärel eelnev tulemuste fail asendatakse.



Joonis 3. Eurostat-Cybernetica projekti raames tehisandmete põhjal loodud soojakaart, mis kirjeldab inimeste ligikaudset asukohta.

2.4.1 Sisendandmed

Töötluse sisendiks on MNO teenuse kasutajate ligikaudsed asukohaandmed esitatud binaarformaadis. Kuna tegemist oli hüpoteetilise prototüübiga, kasutati reaalse andmestiku asemel sünteesitud tehisandmeid. Iga andmerida sisendis sisaldab välju, mis on kirjeldatud tabelis 1.

Tabel 1. MNO teenuse kasutajate asukohaandmete formaat.

Välja nimi	**Tüüp**	**Kirjeldus**
id	16 baiti base64 kodeeringus	Kasutaja pseudonüümitud identifikaator
tile_e	Positiivne täisarv (*uint16*)	Geograafilise ruudustiku koordinaadid ida suunal
tile_n	Positiivne täisarv (*uint16*)	Geograafilise ruudustiku koordinaadid põhja suunal
value_0	Ujukomaarv (*float32*)	Skoori väärtus selles ruudus alamperioodil 0
value_1	Ujukomaarv (*float32*)	Skoori väärtus selles ruudus alamperioodil 1
value_2	Ujukomaarv (*float32*)	Skoori väärtus selles ruudus alamperioodil 2
value_3	Ujukomaarv (*float32*)	Skoori väärtus selles ruudus alamperioodil 3

Geograafilised koordinaadid on kirjeldatud kahe täisarvuga (*tile_e* ja *tile_n*). Võime oletada, et mingi kindel riik on jagatud 1km x 1km ruudustikuks ning igat ruutu saab

kirjeldada unikaalse ida- ja põhjasuunalise koordinaadi kombinatsiooniga.

Lisaks on iga päev jagatud kolmeks perioodiks: hommik, päev ja õhtu. Ujukomaarvud *value_1*, *value_2* ja *value_3* väljendavad ruudu (koordinaadi) "tugevust" vastaval perioodil [27]. Ehk kui näiteks kasutaja veedab hommikul palju aega kontoris, siis on tema kontori koordinaatidel *value_1* väärtus kõrge. Neljas ujukomaarv *value_0* on lihtsalt ülejäänud kolm ujukomaarvu summeerituna. Kõik need ujukomaarvud peavad olema mittenegatiivsed.

Sisuliselt väljendab iga andmerida ligikaudseid koordinaate koos ligikaudse ajavahemikuga, kus MNO teenuse kasutaja on päeva jooksul aega veetnud. Arendatud prototüübis on loodud ka Pythoni skript, millega saab lihtsalt genereerida selliseid andmeid.

3 Jõudlustestide disain ja implementatsioon

Lõputöö peamine eesmärk on luua kaks jõudlustesti ning võrrelda nende töötlusaegasid erinevate sisendite korral. Esimene jõudlustest kasutab Intel SGX-i ja Sharemind HI-d koos Streams teegiga ning teine jõudlustest kasutab Intel TDX-i, Rusti ja Rusti iteraatoreid. Põhjus miks kasutati erinevaid keeli on see, et Intel SGX-i puhul utiliseeriti Sharemind HI-d, mis on suunatud C++ arendusele. Teiselt küljelt kasutati Intel TDX-i puhul Rusti ja Rusti iteraatoreid, sest loodeti, et need pakuvad arendajale kokkuvõttes mugavamat ja kaasaegsemat arenduskogemust kui C++ ja Streams teek. Nende tööriistade võrdlust on võimalik lugeda peatükis 5.

Selles peatükis on kirjeldatud jõudlustestide implementeerimiseks tehtud disainivalikuid koos põhjendustega, miks sellised valikud tehti.

3.1 Jõudlustestide disain

Jõudlustestide loomiseks võeti eeskuju peatükis 2.4 kirjeldatud Eurostati projektist, mis loodi koostöös Cyberneticaga. Autori jõudlustestid on loodud käsitlema samu sisendandmeid, mis eelnevalt mainitud projekt. Samuti imiteeriti andmetöötlusloogika implementeerimisel Eurostati projekti loogikat, kuigi mõnevõrra lihtsustatult.

Põhjus miks kasutati Eurostati projekti eeskujuks on sellepärast, et see võimaldas luua jõudlustestid, mille andmetöötlusloogika on sarnane olukorrale, mis võib esineda reaalmaailmas. Samuti oli võimalik kasutada projektis juba olemasolevat andmete generaatorit sisendi sünteesimiseks, mis võimaldas säästa arendusele kulutatavat aega. Lisaks on Eurostati projektis kirjeldatud mõned Intel SGX-i, Sharemind HI-d ja Streams teeki kasutava jõudlustesti tulemused. Neid saab autor vajadusel kasutada võrdluspunktina selle lõputöö raames mõõdetud tulemustele.

Jõudlustestide eesmärk on mõõta aega, mis kulub kogu andmetöötluse töövoo täitmisele algusest lõpuni. See tähendab, et näiteks lugemis-, kirjutamis- ja arvutusoperatsioonide jõudlust ei mõõdeta eraldi, vaid käsitletakse ühe tervikliku protsessina. Selle lähenemise põhjuseks on asjaolu, et tänapäeva protsessorid on väga keerukad ning toetavad mitmetasandilist rööptöötlust (näiteks *out-of-order execution* ja paralleelne andmete laadimine ning salvestamine [28]). Kui mõõta üksnes töövoo üksikuid osi, jääb palju määramatust selle kohta, kuidas algoritm tegelikkuses käitub. Seetõttu on mõistlik testida algoritmi kui tervikut, kuna see annab usaldusväärsema pildi süsteemi reaalsest jõudlusest. Loomulikult oleks teatud olukordades kasulik mõõta ka üksikute faaside jõudlust, kuid selle

lõputöö kontekstis on tervikliku elulise töövoo mõõtmine probleemipüstituse seisukohalt otstarbekam.

Jõudlustestides implementeeritud loogika on järgnev:

1. Pseudonüümitud sisendandmete failist sisse lugemine. Sisendfailis iga rida näitab infot ühe kasutaja asukoha kohta teatud ajaperioodi(de)l.
2. Andmete depseudonüümimine.
3. Andmete sorteerimine identifikaatori ja koordinaatide järgi kasvavasse järjekorda.
4. Vigaste ja tühjade sisendandmete töötlusest eemaldamine. Kirje eemaldatakse, kui selles:
   a) Esineb ujukomaarv, mis pole lõplik.
   b) Esineb negatiivne ujukomaarv.
   c) Ei esine ühtegi positiivset ujukomaarvu (tühi kirje).
5. Andmetest sama võtmega (identifikaator ja koordinaadid samasugused) kirjete ühendamine, võttes maksimaalse ujukomaarvulise väärtuse iga ajaperioodi kohta.
6. Tulemuse faili kirjutamine:
   a) Kui tegemist on esimese töötlusega, siis luuakse uus fail ning kirjutatakse krüpteeritud tulemus sinna faili.
   b) Kui tulemuste fail on juba olemas (st pole esimene töötlus), siis toimub järgmine protsess:
      i) Loetakse sisse ja dekrüpteeritakse eelnev tulemuste fail.
      ii) Eelnevad tulemused ühendatakse uue sisendiga (*outer join*), kusjuures andmed jäävad sorteerituks.
      iii) Duplikaatvõtmetega kirjed ühendatakse, summeerides ujukomaarvud iga ajaperioodi kohta.
      iv) Lõpuks kirjutatakse tulemuste fail uuendatud andmetega krüpteeritud kujul üle.

Jõudlustestide loogikat kirjeldav UML (*Unified Modeling Language*) skeem on lisas 2.

Tulemuste faili krüpteerimiseks kasutatakse AES-GCM algoritmi [29], kuna see on Intel SGX-i poolt toetatud. Pseudonüümimisloogikas kasutati Eurostati projekti eeskujul AES-CTR algoritmi [30].

Töötlusloogika kirjeldusest on selge, et iga töötlusega andmete arv kasvab ning jooksutamiseks kuluv aeg suureneb, kuna uued andmed ühendatakse juba olemasolevate andmetega.

Selline loogika on implementeeritud nii Intel SGX-i lahenduses C++-i, Sharemind HI ja Streams teegiga kui ka Intel TDX-i lahenduses Rusti ja Rusti iteraatoritega. Lahendused

on proovitud teha võimalikult sarnased ilma keelelisi mugavusi ohverdamata.

Jõudlustestide arendamiseks loodi Giti koodihoidla, mis sisaldab mõlemat jõudlustesti ning skripte, millega neid jooksutada genereeritud andmetega. Andmete formaat on kirjeldatud tabelis 1. Koodihoidla on tehtud avalikuks ning on leitav lisas 3.

3.1.1 Ühelõimeline lähenemine

Mõlemad jõudlustestid on ühelõimelised programmid. Selline lähenemine võimaldab keskenduda usaldatud täitmiskeskkonna (Intel SGX või Intel TDX) enda töötluskulu võrdlemisele ilma täiendava keerukuseta, mida toovad kaasa mitmelõimelised töövood. Lisaks oli Eurostati projekt lihtsuse huvides ja Sharemind HI piirangu tõttu samuti ühelõimeline.

Siiski tasub märkida, et juhul kui algoritm on lihtsasti paralleliseeritav ning süsteemil saab mitut protsessorituuma kasutada, siis sellisel juhul on mitmelõimelisuse rakendamine lihtsam Intel TDX-i lahenduse puhul, kuna Sharemind HI ei toeta enklaavis mitme lõime kasutamist. Lisaks on Intel TDX-i kasutamisel mitmelõimelisus suurema tõenäosusega turvaline (vt peatükk 2.1.2).

3.1.2 Jõudlustestide jooksutamine

Jõudlustestide jooksutamiseks on loodud koodihoidlas Bash skriptid. Bash skriptile saab sisendina lisada parameetrid, mille järgi genereeritakse automaatselt sisendandmete failid. Samuti saab täpsustada, mis keskkonnas jooksutada jõudlustesti (Intel SGX enklaav või Intel TDX VM). Skriptid on loodud selleks, et vähendada manuaalset käskude sisestamist ning seekaudu lihtsustada jõudlustestide täitmisprotsessi ja tagada korduvust.

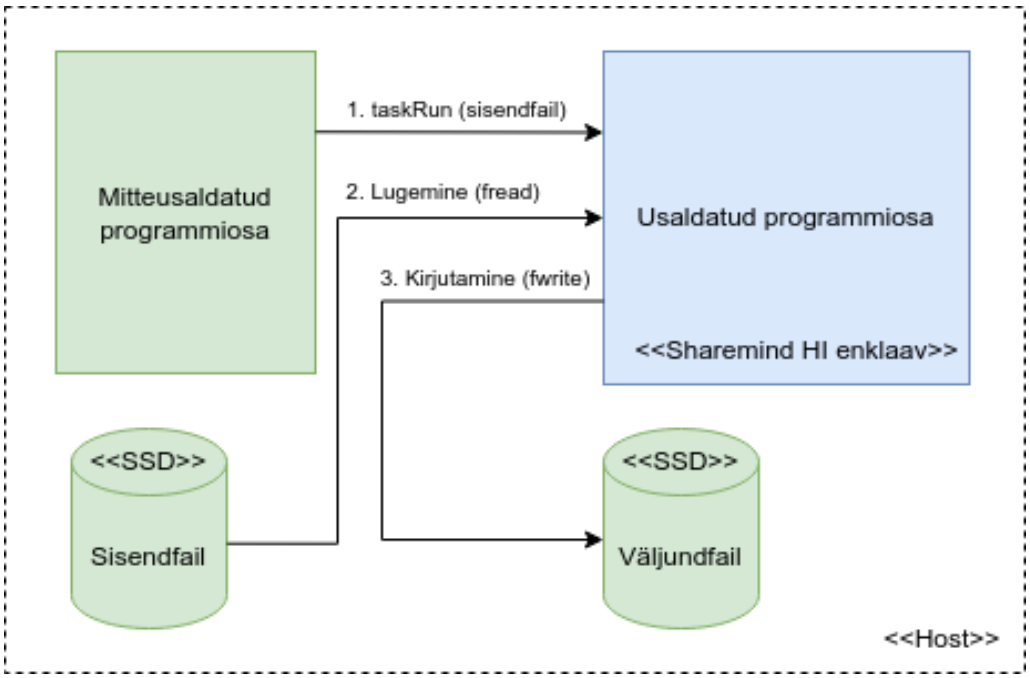
Jõudlustestide jooksutamiseks vajalike sõltuvuste hankimiseks kasutatakse spetsiaalset Nixiga loodud töökeskkonda [31], [32]. See on konfiguratsioonifaili abil kirjeldatud keskkond, kus on vajalikud sõltuvused olemas, ilma et arendaja peaks need ükshaaval oma seadmele alla laadima ja paigaldama. Nixi kesta sisenemine võimalik ühe lühikese käsuga. Seetõttu lihtsustab Nixi kesta kasutamine märgatavalt arendusprotsessi juhul, kui projekti arendamiseks või jooksutamiseks on vaja arvukalt sõltuvusi ning arendustööd tehakse mitmel eri seadmel. Kuna projekti arendamine toimus enamasti lokaalselt autori arvutis, aga jõudlustestide jooksutamine kaugseadmetel, siis Nixi kasutamine lihtsustas oluliselt projekti algset ülesseadmist ning uute sõltuvuste lisamist. Samuti saab Nixiga kindlustada, et iga sõltuvus omab samat versiooni mõlemal seadmel. Kasutatud Nixi kesta konfiguratisoonifail on leitav lisas 4.

3.2 Jõudlustesti implementatsioon Intel SGX-i jaoks

Intel SGX-i lahenduse jaoks kasutati jõudlustesti arendamisel Sharemind HI-d ning Streams teeki. Keeleks kasutati C++-i. Kuna samu tööriistu kasutatakse ka Eurostati projektis, sai autor jõudlustesti loomisel taaskasutada mitmeid selle projekti avalikus koodihoidlas [25] olevaid komponente. Siiski tuli võrdlemisi suur osa lahendusest ka iseseisvalt luua.

Intel SGX enklaavis jooksev kood on disainitud viisil, et andmete töötlemine ei hõivaks korraga suurt kogust mälu. Sellega aitas Streams teek, mis võimaldab nii-öelda laisalt andmeid töödelda. Intel SGX jõudlustesti põhiloogika on leitav lisas 5.

Jõudlustesti saab jooksutada vaid seadmel, millel on Intel SGX-i toetav protsessor [33]. Lisaks peab arvutis olema üles seatud Sharemind HI ja töötav Nixi paigaldus. Jooksutamine on tänu Bash skriptidele lihtne: piisab Nix kestas ühe käsu sisestamisest. Skript genereerib sisendandmed, käivitab Intel SGX enklaavi ning täidab selles andmetöötlusprotsesse kuniks kõik sisendfailid on töödeldud ja ühendatud väljundfaili. Lahendust kirjeldab lihtsustatud kujul skeem joonisel 4.



Joonis 4. Intel SGX-i jõudlustesti implementatsioon.

Tuleb märkida, et Intel SGX jõudlustestide käivitamisel ei ole rakendatud LVI (*Load Value Injection*) vastaseid leevendusi [34].

3.3 Jõudlustesti implementatsioon Intel TDX-i jaoks

Intel TDX-i lahenduse jaoks kasutati jõudlustesti arendamisel Rusti ja Rusti iteraatoreid, et võrrelda, kas Rust parandab arenduskogemust võrreldes C++-i ja Streams teegiga. Rust programmis ei pidanud ühtegi TEE-spetsiifilist teeki kasutama, kuna Intel TDX võimaldab teoorias käivitada mistahes programme ilma koodi kohandamata.

Lahenduses kasutati teeke nagu Itertools (vt peatükk 2.3.1), et laiendada standardteegi iteraatorite funktsionaalsust ning Bytemuck [35], et lihtsustada binaarfaili sisselugemis- ja kirjutamisloogikat. Lisaks kasutati mitmeid teeke pseudonüümimis- ja krüpteerimisloogika implementeerimiseks.

Sarnaselt Intel SGX-i lahendusele töödeldakse andmeid laisalt, et mitte hõivata korraga liigselt suurt kogust mälu. Intel TDX jõudlustesti põhiloogika on leitav lisas 6.

Intel TDX-i jõudlustesti jooksutamine on mõnevõrra keerulisem kui Intel SGX-iga, kuna lisandub virtuaalmasina element. Seda jõudlustesti on võimalik jooksutada vaid seadmetel, millel on Intel TDX-i toetav protsessor [36]. Lisaks peab arvuti kasutama kindla versiooniga Linux kernelit, mis on spetsiaalselt konfigureeritud ja paigatud (ingl *patched*) Intel TDX-i jaoks [37]. Samuti peab seadmes olema ehitatud TDVF kujutis [38] ning alla laetud Intel TDX-iga ühilduv QEMU versioon VM-i emuleerimiseks [39], [40].

Lisaks tuleb luua ka virtuaalmasina kujutis, mis kasutab Intel TDX-iga ühilduvat kernelit. Autor katsetas VM-i jaoks Ubuntu ja Debian operatsioonisüsteeme (ingl *Operating System*, OS) ning mõlemad töötasid Intel TDX-iga. Need OS-id valiti testimiseks, kuna nende seadistamine on hästi dokumenteeritud ning need on aktiivselt toetatud. Lisaks lõi autor ka Nixi konfiguratsioonifaili, mis loob automaatselt NixOS-i [41] kasutava virtuaalmasina konfigureeritud ja paigatud kerneliga.

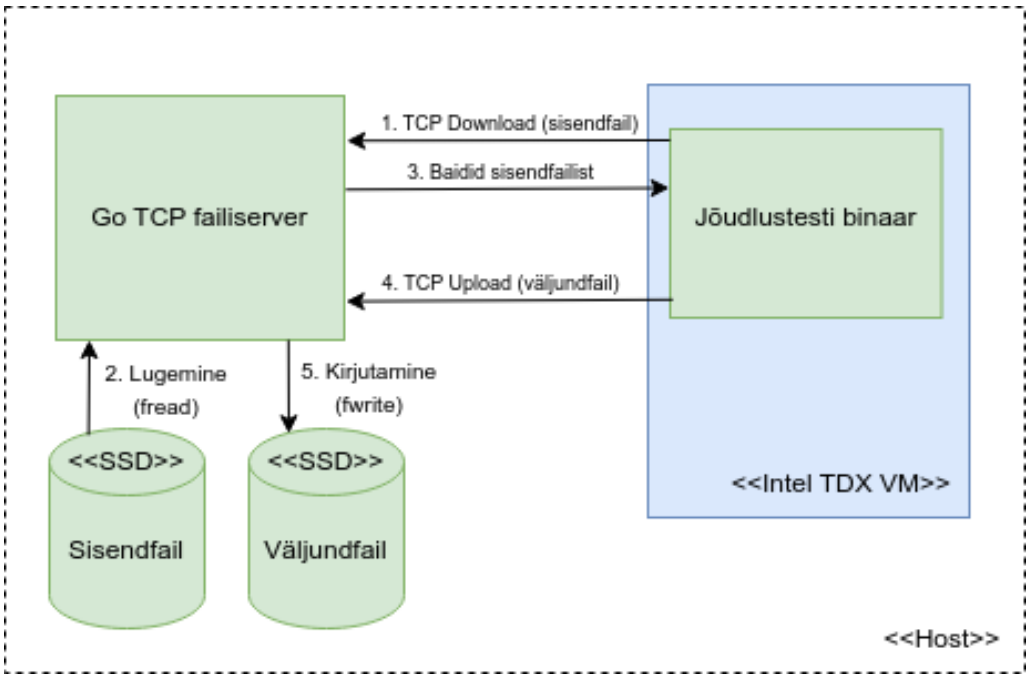
Kuid NixOS VM-i käivitamine viis krahhini, kui mälukasutus ületas 2 GB. Täpset juurpõhjust ei suutnud autor välja selgitada. Sarnased probleemid võivad Intel TDX-i puhul olla tingitud sellest, et isegi kui Intel TDX-i riistvara on mitmetel protsessoritel olemas, on Intel TDX-i toetav tarkvaraline ökosüsteem siiski algfaasis. Näiteks pole Intel TDX-i toetus siiani täielikult Linuxi kernelisse lisatud ning seetõttu tuleb kasutada paigatud kernelit. Algfaasis ökosüsteemi tõttu võivad esineda ootamatud vead ja stabiilsusprobleemid.

NixOS-iga tekkinud vea tõttu otsustas autor jõudlustestide jooksutamiseks kasutada Ubuntu operatsioonisüsteemi, järgides seadistamisel ettevõtte Canonical poolt loodud juhendit [40]. VM kasutas modifitseeritud Linux kernelit versiooniga 6.8.0 ning Ubuntu OS-i versiooniga 22.04.

Siiski oli Nixi kasutamine VM-i loomiseks mugav, kuna kogu protsess oli täielikult automatiseeritud ning Nixi konfiguratsioonifailis sai täpsustada kõiki programme ja sõltuvusi, mis peaksid VM-is olema. Sellepärast oleks tulevikus kasulik uuesti kaaluda Nixi kasutamise varianti ning proovida esinenud mäluprobleem lahendada.

Lisaks ilmnes Intel TDX-i lahenduse implementeerimisel probleem sisend- ja väljundfailide käsitlemisel. Algselt kasutas autor 9p failisüsteemi [42], mille abil loodi hosti ja virtuaalmasina vahel mauntitud kaust (ingl *mount*) andmevahetuseks. See lahendus osutus aga aeglaseks ega sobinud suuremahuliste andmete töötlemiseks. 9p kasutamisel muutusid IO operatsioonid kitsaskohaks, mistõttu suurenes töötlusaeg kuni kolmekordselt.

Kuna Intel TDX-i VM-ide puhul ei paistnud olevat tuge alternatiivsete failisüsteemide kasutamiseks (või vähemalt ei õnnestunud autoril neid edukalt seadistada), otsustati valida teistsugune lähenemine. Autor lõi Go-põhise TCP (*Transmission Control Protocol*) failiserveri, mis võimaldas hosti ja VM-i vahel andmeid vahetada. Go valiti keele hea jõudluse tõttu, ning TCP kasutamine HTTP asemel võimaldas pisut efektiivsemat andmevahetust[1]. Server loodi viisil, mis võimaldas puhvrite abil failist lugemist ja faili kirjutamist. Lisaks tuli kohandada virtuaalmasina võrguliidest, et tagada kiirem suhtlus hosti ja VM-i vahel. Kokkuvõttes osutus see lahendus IO-kiiruse seisukohalt kõige tõhusamaks. Lahendust kirjeldab lihtsustatud kujul skeem joonisel 5.



Joonis 5. Intel TDX-i jõudlustesti implementatsioon.

[1]Väide põhineb autori tehtud eksperimendil, kus TCP osutus lokaalselt teostatud andmevahetusolukorras veidi kiiremaks kui HTTP.

Jõudlustestide käivitamine toimub sarnaselt Intel SGX-i lahendusele, kasutades Nixi kestakeskkonda. Erinevuseks on see, et Bash skript käivitab VM-i, mis jooksutab automaatselt systemd teenuse [43] abil andmetöötlusprotsesse kuniks kõik andmed on töödeldud. Seejärel VM-i protsess peatatakse.

4 Jõudlustestide tulemused

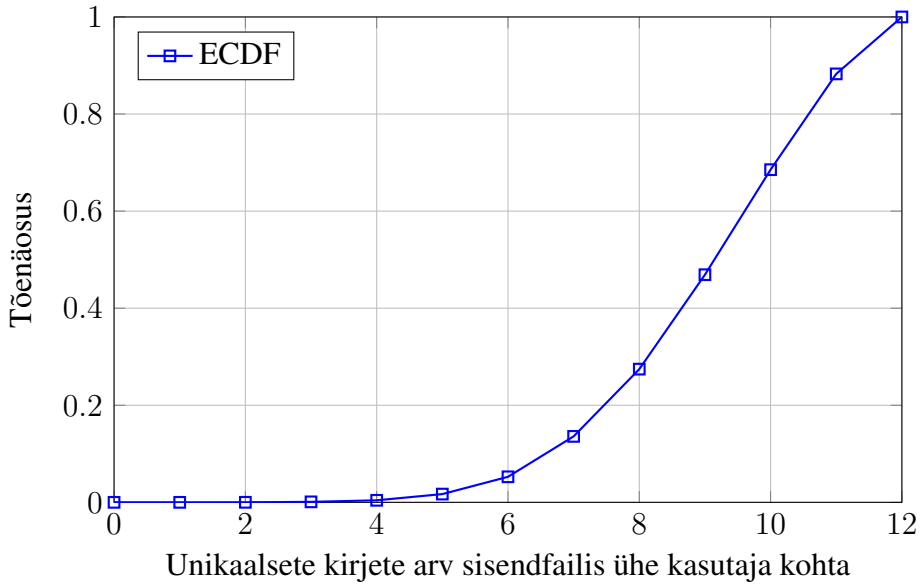
Jõudlusteste jooksutati sünteesitud andmetega, mis genereeriti kasutades Eurostati projekti koodihoidlas leitavat andmegeneraatorit [25]. Kahe jõudlustesti puhul töödeldi 90 sisendfaili. Eurostati poolt välja mõeldud hüpoteetilises MNO andmetöötlusolukorras luuakse iga päeva lõpuks uus sisendfail, mis sisaldab selle päeva andmeid. Seega tähendab 90-päevane periood sisuliselt 90 sisendfaili töötlemist. Ülejäänud jõudlustestide jaoks kasutati vähem sisendfaile, kuna 90 faili kasutamisel oleks saanud kettaruum otsa.

Mõlema lahenduse puhul (Intel SGX koos Sharemind HI ja Streams teegiga ning Intel TDX koos Rusti iteraatoritega) jooksutati jõudlusteste samasuguste andmetega. Kõik viis testimismalli on kirjeldatud tabelis 2.

Tabel 2. Testimismallid ja jõudlustestide tulemused.

		Ühik	**Test1**	**Test2**	**Test3**	**Test4**	**Test5**
Parameeter	Periood	Päev	90	90	60	30	20
	Kasutajate arv	Miljon	1	5	10	20	30
	Sisendfaili suurus	GB	0.3	1.6	3.3	6.6	9.9
	Tulemuse faili suurus	GB	24	122	165	174	175
Intel SGX	Pikim töötlusaeg päevadest	Minut	1.6	8.3	11.6	13.3	13.8
	Töötlusaeg kokku	Minut	74	**381**	**363**	**217**	**159**
Intel TDX	Pikim töötlusaeg päevadest	Minut	1.5	7.7	9.4	10.2	10.0
	Töötlusaeg kokku	Minut	62	**330**	**284**	**159**	**119**

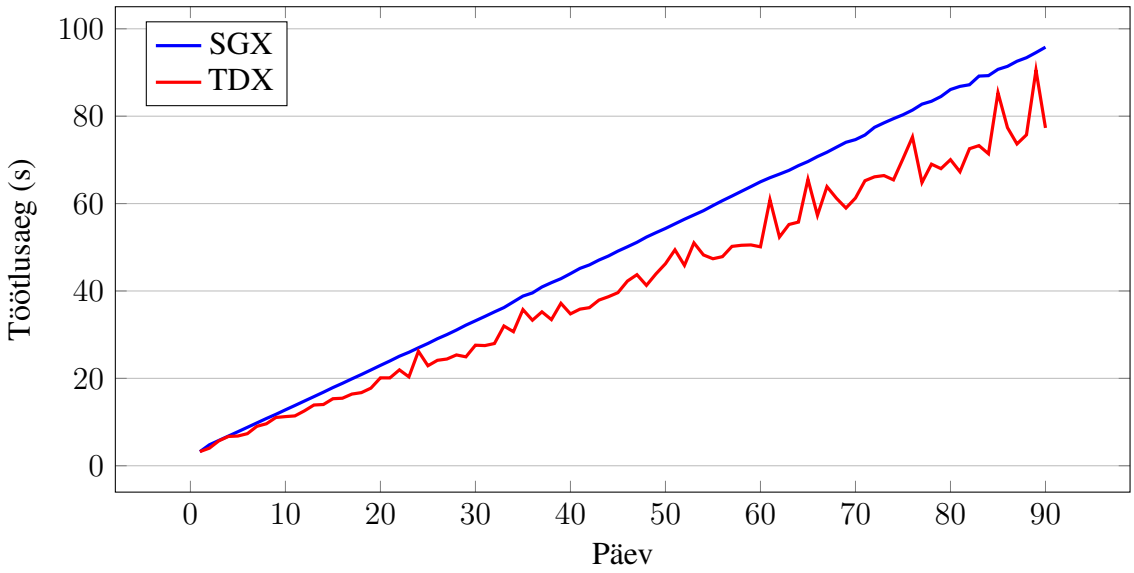
Perioodi parameeter kirjeldab, mitut sisendfaili töödeldakse. Kasutajate arvu parameeter kirjeldab mitu unikaalse identifikaatoriga kirjet on igas sisendfailis, kusjuures iga unikaalse kasutaja kohta esineb suvaline arv kirjeid. Kirjete arvu kasutaja kohta näitab empiiriline kumulatiivne jaotus joonisel 6. Graafiku Y-teljel on näidatud tõenäosus, et kasutajal on nii mitu kirjet või vähem ning X-teljel unikaalsete kirjete arv sisendandmetes kasutaja kohta. Keskmiselt on sisendis 9 unikaalset kirjet MNO teenuse kasutaja kohta ning kirjete arv on vahemikus kahest kaheteistkümneni.



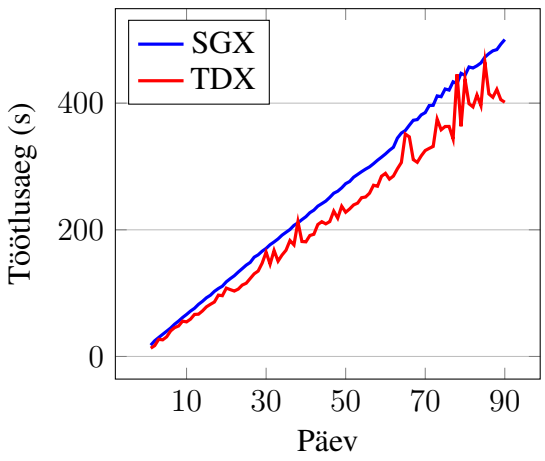
Joonis 6. Sisendfaili kirjete arvu empiiriline kumulatiivne jaotusfunktsioon (ECDF) ühe unikaalse kasutaja kohta.

4.1 Mõõdetud tulemused

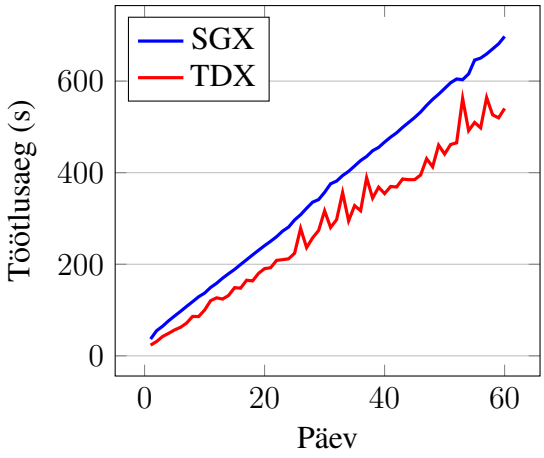
Mõõdetud töötlusaegasid on näha joonistel 7, 8, 9, 10 ja 11. Iga graafik väljendab nii Intel SGX-i kui ka Intel TDX-i lahenduse töötlusaegasid ühe testimismalli kohta tabelist 2.



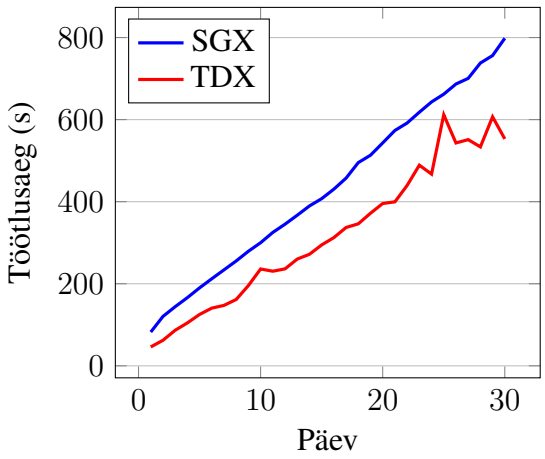
Joonis 7. Intel SGX ja Intel TDX töötlusajad esimese testimismalli puhul (miljon kasutajat).



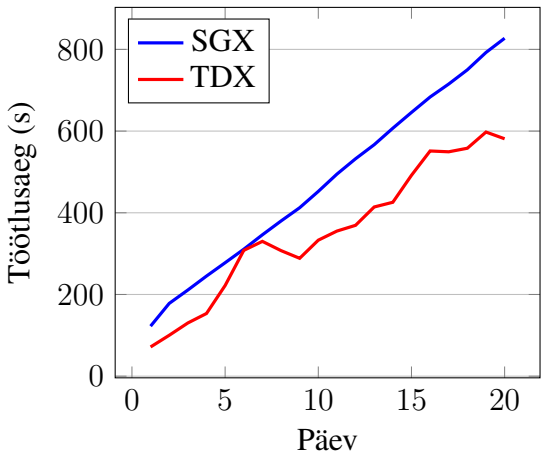
Joonis 8. Intel SGX ja Intel TDX töötlusajad teise testimismalli puhul (5 miljonit kasutajat).



Joonis 9. Intel SGX ja Intel TDX töötlusajad kolmanda testimismalli puhul (10 miljonit kasutajat).



Joonis 10. Intel SGX ja Intel TDX töötlusajad neljanda testimismalli puhul (20 miljonit kasutajat).



Joonis 11. Intel SGX ja Intel TDX töötlusajad viienda testimismalli puhul (30 miljonit kasutajat).

4.2 Tulemuste analüüs

Eelnevatelt joonistelt on näha, et Intel TDX lahendus on üldiselt kiirem kui Intel SGX lahendus, mis on olemasolevaid jõudlusanalüüse arvestades oodatav tulemus [44]. Tabelis 3 on näidatud iga testimismalli puhul lahenduste keskmised protsentuaalsed erinevused sama päeva töötlusaegade vahel ning protsentuaalsete erinevuste standardhälve.

Tabel 3. Intel SGX ja Intel TDX lahenduste töötlusaegade keskmine erinevus ja standardhälve.

	Test1	**Test2**	**Test3**	**Test4**	**Test5**
Keskmine protsentuaalne erinevus sama päeva töötlusaegade vahel (%)	17.3	15.4	27.2	34.9	29.8
Protsentuaalsete erinevuste standardhälve päevade lõikes (%)	6.0	5.9	8.6	10.7	13.1

Tabelist 3 on näha, et viie testimismalli puhul on Intel TDX-i lahendus 15% kuni 35% kiirem kui Intel SGX-i lahendus. Tundub, et erinevus töötlusaegades suureneb koos andmehulgaga, mis võib viidata paremale skaleeritavusele Intel TDX-i lahenduse puhul. Samas suureneb ka protsentuaalsete erinevuste standardhälve, mis viitab suuremale varieeruvusele Intel TDX-i jõudluses suuremate andmemahtude korral.

Intel SGX-i lahendus kirjutab suurte andmemahtude sorteerimisel vahetulemused kettale (Sharemind HI implementatsiooni tõttu), samas kui Intel TDX-i puhul toimub sorteerimine mälus. Vahetulemuste salvestamine kettale suurendab tõenäoliselt töötlusaega, kuna ketta kasutamine on oluliselt aeglasem kui mälupõhine töötlus.

Siiski sorteeritakse ainult sisendfail, mille suurus on igas testimismallis konstantne. Seetõttu isegi kui sorteerimise ajakulu erineb kahe lahenduse vahel märkimisväärselt, ei tohiks see ajakulu päevade lõikes kasvada. Ehk sisendfaili korduv sorteerimine ei selgita, miks jõudlustestide ajakulu laheneb üha rohkem, nagu näha ka graafikutelt.

Lisaks peaks mainima, et teoorias on Rusti iteraatoreid kasutades väga lihtne paralleliseerida ühelõimelist töötlust kasutades Rayon teegi `par_iter` funktsionaalsust [45]. Sellega saab mitmel juhul olemasoleva Rusti standardteegi iteraatorite loogika paralleliseerida ilma suuri muudatusi tegemata. Peale lühidat katsetamist suutis autor Intel TDX-i lahenduse sorteerimis- ja filtreerimisloogika paralleliseerida. Samas muu loogika jäi muutmatuks, kuna Itertools teek ei ühildu Rayoni funktsionaalsusega. Samuti ei olnud võimalik paralleliseerida depseudonüümimisloogikat ilma suuremaid koodimuudatusi tegemata.

5 Arenduskogemuse võrdlus

See peatükk võrdleb arenduskogemust mõlema lahenduse loomisel. Peamiselt võrreldakse keelelisi erinevusi Rusti ja C++-i vahel ning Rusti iteraatoreid ja Streams teeki. Puudutatakse teemasid nagu kompilaatori veateated, trafarettkoodi kogus, andmetöötlusloogika implementeerimine Streams teegi ja Rusti iteraatorite abil ning õppimisprotsessi keerukus.

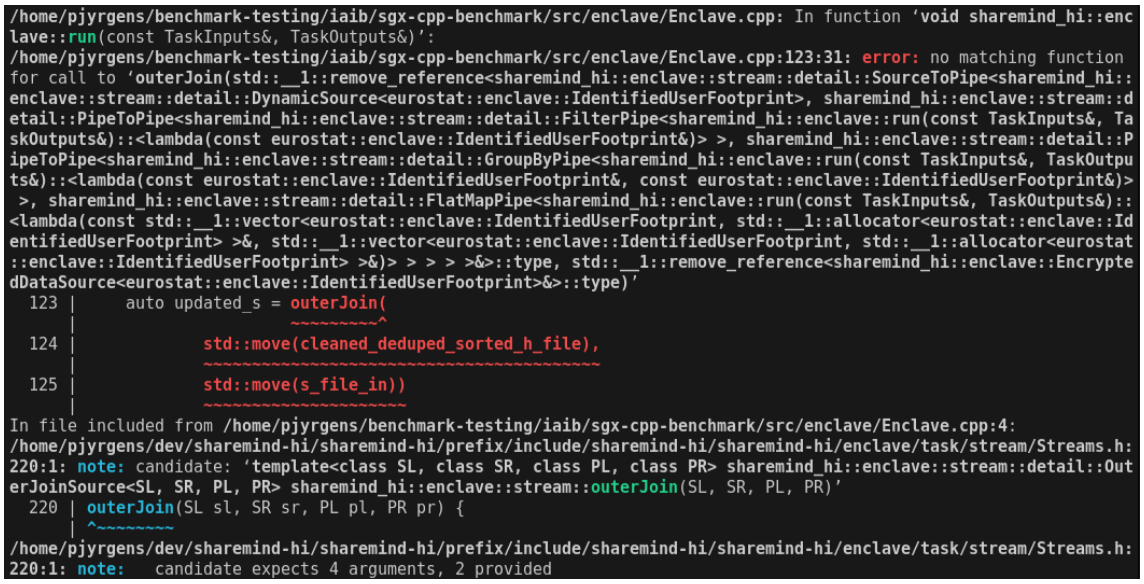
5.1 Kompilaatori veateated

C++ puhul on võimalik valida mitme kompilaatori vahel, millest igal on oma tugevused ja nõrkused olenevalt kasutusjuhust [46]. Intel SGX-i lahenduse arendamisel kasutas autor GNU kompilaatorite kollektsiooni (ingl *GNU Compiler Collection*, GCC) [47], kuna see on ainus C++ kompilaator, mida Intel SGX SDK ametlikult toetab Linux operatsioonisüsteemidel [48]. Teised populaarsed C++ kompilaatorid on näiteks Microsoft Visual C++ Compiler [49] ning Clang [50].

GCC kompilaatori veateated on tuntud oma mahukuse ja keerukuse poolest, mis muudab need arendajatele sageli raskesti mõistetavaks [51]. Ka lihtsamate süntaksivigade puhul koodis võib kompilaator näidata nimekirja mitmekümnest veateatest ja hoiatusest, millest arendaja peab leidma vea juurpõhjuse või -põhjused.

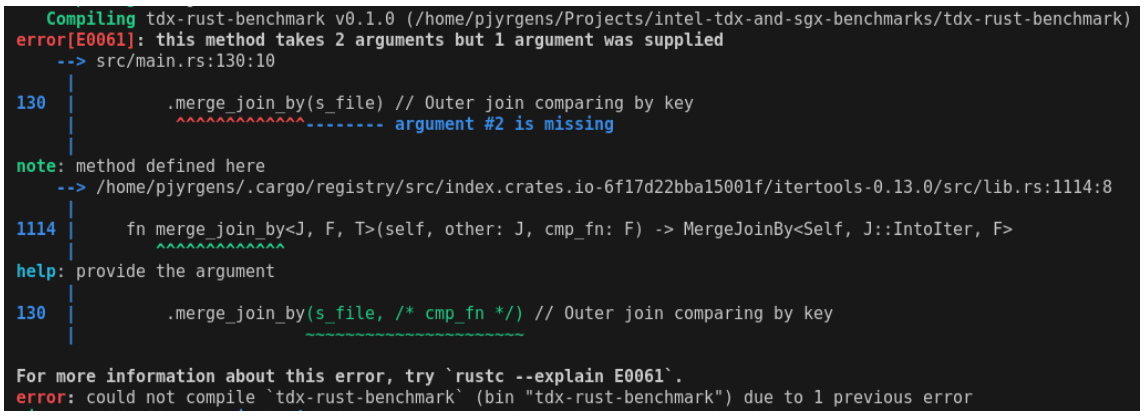
Sageli põhjustab üksainus viga terve rea teisi veateateid, mis ei pruugi otseselt viidata algsele probleemile. See juhtub tavaliselt seetõttu, et hilisemad koodiread sõltuvad varasematest, kus võis esineda näiteks süntaksiviga. GCC kompilaator ei piirdu ainult algse vea kirjeldamisega, vaid väljastab ka kõik sellest tulenevad vead, mis suurendab veateadete hulka ja võib muuta probleemi leidmise keerukamaks. Joonisel 12 on toodud näide GCC veateatest, mis tekkis Streams-teegi abimeetodi ebakorrektse kasutamise tõttu.

Teiselt küljelt on GCC kompilaatoriga võimalik saada ka väga lakoonilisi veateateid, millel puudub viide konkreetsele põhjusele, miks see tekkis [52]. Kuigi sellised veateated on haruldased, võib nende lahendamine olla väga keeruline, kuna vea tekkepõhjus on tundmatu.



Joonis 12. GCC veateade Streams teegi abimeetodi ebakorrektsel kasutamisel.

Rust on seevastu tuntud oma selgete ja kasutajasõbralike veateadete poolest [53]. Rusti kompilaatori veateated järgivad pritntsiipi näidata kasutajale võimalikult vähe infot nii, et see oleks piisav probleemist aru saamiseks ja selle lahendamiseks. Vea asukoht on reeglina täpselt kirjeldatud ning sageli on kaasas ka soovitused selle parandamiseks. Lisaks on Rusti veateated enamasti rohkem inimloetavad. Soovi korral on võimalik kasutada ka `-explain` lippu, et näha rohkem detaile vea kohta. Joonisel 13 on võimalik näha Rusti kompilaatori veateadet iteraatorite abimeetodi ebakorrektsel kasutamisel.



Joonis 13. Rusti kompilaatori veateade iteraatorite abimeetodi ebakorrektsel kasutamisel.

Võrreldes GCC kompilaatori veateadetega olid Rusti kompilaatori omad selgemad ning lühemad. Selle tulemusel kulus Rusti puhul enamasti vea ilmnemisel vähem aega selle tuvastamiseks ja parandamiseks. GCC puhul oli sama tüüpi vigade leidmisele kuluv aeg mõnel juhul kordades pikem[1]. Kuigi GCC mahukus võib teatud olukordades olla kasulik, ei tekkinud Intel TDX-i lahenduse loomisel autoril olukordi, kus Rusti veateadete

[1]Subjektiivne hinnang, mis ei põhine otsestel mõõtmistel.

lakoonilisus oleks muutunud takistuseks. Intel SGX-i lahenduse loomisel seevastu oli mitmeid olukordi, kus kompileerimisel tekkis viga, mille juurpõhjus ei olnud veateatest autorile arusaadav.

Tuleb märkida, et C++20 standard tutvustas *concepts*-nimelist funktsionaalsust [54], mis võimaldab kompileerimisvigade korral anda kasutajasõbralikumaid ja täpsemaid veateateid. Kahjuks ei toeta Intel SGX SDK hetkel veel C++20 standardit ning piirdub C++17 versiooniga, mistõttu *concepts* ei olnud jõudlustesti loomisel kasutatav.

5.2 Trafarettkood

Trafarettkood (ingl *boilerplate code*) on kood, mis kordub erinevates kohtades ilma märkimisväärsete muutusteta. Üldiselt on hea harjumus võimalusel vältida trafarettkoodi, et hoida programmikood kompaktsem ja sageli ka paremini loetav.

C++ nõuab sageli rohkem trafarettkoodi kirjutamist kui Rust. Üheks põhjuseks on C++-is väljakujunenud tava kasutada päisefaile (ingl *header files*) [55], mis suurendab korduva koodi mahtu. Päisefailide eesmärk on eraldada funktsioonide, klasside ja objektide deklareerimine nende tegelikust implementatsioonist, mis aitab parandada koodi hooldatavust ja võimaldab nende korduvkasutust teistes programmifailides. Ühtlasi tähendab see arendaja jaoks täiendavat tööd ja lisakoodi kirjutamist.

Samas Intel SGX-i lahenduse arendamisel sai mitmel juhul vältida trafarettkoodi kirjutamist tänu Streams teegile, mis abstraheeris palju keelelist keerukust. Lisaks tuleb märkida, et tulevikus võib C++ moodulite (*modules*) [56] laiem kasutuselevõtt aidata vähendada vajadust päisefailide järele.

Rusti puhul seevastu ei ole vaja eraldi päisefaile luua, mis vähendab korduva koodi kirjutamist. Lisaks pakub Rust mitmeid modernseid keelelisi mugavusi, nagu tüübijäreldus (ingl *type inference*), *trait'i* süsteem ja tühiväärtuste kontrollimine (ingl *null checking*), mis kõik aitavad vähendada trafarettkoodi.

Intel TDX-i lahenduse teostamiseks kirjutas autor 525 rida Rust koodi. Intel SGX-i lahenduse teostamiseks kirjutas autor 721 rida C++ koodi, mis on 37% rohkem võrreldes Rusti lahendusega. Koodiridade mõõtmiseks kasutas autor Tokei tööriista [57].

Siiski tuleb arvestada, et koodiridade arv iseenesest ei ole objektiivne mõõdik trafarettkoodi koguse hindamiseks. See sõltub paljudest teguritest, nagu keele süntaktilised eripärad, koodi struktuur ning arendaja isiklik programmeerimisstiil. Seetõttu ei saa pelgalt koodiridade põhjal teha täielikult objektiivseid järeldusi ühe või teise lahenduse paremusest.

5.3 Streams teegi ja Rusti iteraatorite võrdlus

Streams teegi kasutamine andmete töötlemiseks oli võrdlemisi lihtne ja mugav. Teegis olid olemas kõik abimeetodid, mida autor soovis kasutada ning nende meetodite kokku aheldamine ei tekitanud probleeme. Siiski osutus keeruliseks õigete meetodite leidmine ning nendest aru saamine, kuna laialdast dokumentatsiooni teegi funktsionaalsuse jaoks ei ole tehtud. Seetõttu pidi korduvalt uurima teegi sisu, et aru saada, mida mõni abimeetod täpsemalt teeb. Samuti kuna tegemist on Cybernetica enda poolt arendatud teegiga, siis ei leidu internetis ressursse probleemide lahendamiseks. Joonisel 14 on näide Streams teegi laiendamisest.

```
// Define a new Stream extension to filter even numbers
namespace stream {
    struct KeepEven {
        bool operator()(int32_t x) const {
            return x % 2 == 0;
        }
    };

    /// Extend the Streams library with a pipeable function
    ↪ `filterEven`
    inline detail::FilterPipe<KeepEven> filterEven() {
        return filter(KeepEven{});
    }
}

void run(TaskInputs const & inputs, TaskOutputs & outputs) {
    // Use filterEven on some arbitrary data.
    auto evenNumbers = stream::range(1, 9)
                     >>= stream::filterEven()
                     >>= stream::collect();

    // Print result
    for (int32_t num : evenNumbers) {
        std::cout << num << " ";
    }
}
```

Joonis 14. Streams teegi laiendamise koodinäide.

Rusti iteraatorid on laialdaselt kasutusel ja hästi dokumenteeritud. Siiski ei sisaldanud Rusti standardteek kogu vajalikku funktsionaalsust, mida autor soovis kasutada. Seetõttu tuli lisaks kasutada ka Itertools teeki. Õnneks on ka selle teegi funktsionaalsus põhjalikult dokumenteeritud. Kasutusjuhendite ja näidete olemasolu pärast oli Rusti iteraatorite ja Itertools teegi kasutamine mõnevõrra lihtsam kui Streams teegi kasutamine. Näide Rusti iteraatorite laiendamisest on joonisel 15.

```rust
trait FilterEven: Iterator<Item = i32> {
    fn filter_even(self) -> std::iter::Filter<Self, fn(&i32) -> bool>
    where
        Self: Sized,
    {
        self.filter(|x| x % 2 == 0)
    }
}

// Implement the trait for all iterators over `i32`
impl<T: Iterator<Item = i32>> FilterEven for T {}

fn main() {
    let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8];

    let evens: Vec<_> = numbers.into_iter().filter_even().collect();

    println!("{:?}", evens);
}
```

Joonis 15. Rusti iteraatorite standardteegi laiendamise koodinäide.

5.4 Õppimisprotsessi keerukus

Vaatamata mõndadele erinevatele keelelistele disainiotsustele võib C++-i ja Rusti õppimiskõveraid (ingl *learning curve*) pidada autori kogemusel mõnevõrra sarnaseks. Mõlemad keeled tutvustavad keerukaid paradigmasid, mis nõuavad uuel arendajal harjumist.

C++ puhul peab arendaja hästi hoomama, kuidas arvuti mälu kasutab, kuna see võimaldab käsitsi mälu hallata ning kasutada viitasid. Algajatel, kes pole varem madalama tasemega keeltega (ingl *low-level language*) kokku puutunud, nõuab see vabadus sageli pikka harjumisperioodi, kuna mälukasutus on delikaatne protsess ning selle mõistmine ei ole autori hinnangul lihtne.

Samuti kuna C++ on tagasiühilduv varasemate standarditega, toetab see mitmeid aegunud mustreid ja lähenemisviise, mida tänapäeval ei peeta enam optimaalseks või turvaliseks. Arendaja peab ise teadma, milline lähenemine on mõistlik ja milline mitte. Näiteks joonisel 16 välja toodud koodinäide kompileerub veatult. Kuid koodi jooksutamisel prinditakse konsooli "*This should not print*", mis võib uuele C++ arendajale jääda arusaamatuks. Sellistele vigadele viidatakse väljendiga "ajas rändamine" (ingl *time travel*) ning joonisel 16 näidatud koodis tekib see tühja osuti poolt viidatud väärtuse kasutamise tõttu. Kui eemaldada koodist kaheksas rida, siis ei prindita konsooli midagi.

```
__attribute__((noinline))
void foo(int* data) {
  bool isData = false;
  if (data) isData=true;
  if (isData) {
    std::cerr << "This should not print\n";
  } else {
    *data = 7; // <- problematic line, dereferencing null pointer
  }
}

int main() {
  foo(nullptr);
}
```

Joonis 16. Näide keerulisest kompilaatori käitumisest C++ koodi puhul.

Pika kasutusajaloo jooksul on C++ muutunud funktsioonirikkaks keeleks, kuid see toob kaasa suure hulga keerulist ja hajutatud informatsiooni, mis teeb õppimise keeruliseks. Heaks näiteks on funktsioon `std::visit`, mis võimaldab ühes muutujas hoida eri tüüpi väärtusi. Näiteks üks muutuja võib olla kas täisarv või tõeväärtus, kuid mitte mõlemad korraga. Idee on iseenesest lihtne ning sellist tüüpilist kasutusmustrit saab teiste keelte, nagu Rusti või Haskelli, puhul rakendada otse ja mugavalt. C++-is on aga selle realiseerimine keeruline: see eeldab mitmete spetsiifiliste ja tihti raskesti mõistetavate keelemustrite tundmist [58].

Rusti õppimist teeb seevastu keeruliseks sellele omane *ownership* mudel, mis kehtestab reeglistiku, kuidas Rust arvuti mälu kasutab. Vaatamata sellele, et arendaja ei pea enam käsitsi mälu haldama, peab ta ikkagi üksikasjalikult mõistma, kuidas Rust seda teeb. See reeglistik võib algajale olla raskesti hoomatav, kuna tutvustatakse uusi kontseptsioone nagu laenamine [59] ja eluajad (ingl *lifetimes*) [60].

Keeli võrreldes on Rusti õppimiskõver autori arvates laugem. Kuigi arendaja peab ikka hästi hoomama, kuidas arvuti mälukasutus toimib, aitavad Rusti ranged kompileerimisaja kontrollid vältida tüüpvigu lihtsasti loetavate veateadetega. See võimaldab kiiresti õppida oma vigadest. C++ on keelena vabam, kuid seetõttu on näiteks võimalik kompileerida koodi, mis ei käsitle mälu turvaliselt. Reeglina kui Rusti kood kompileerub, siis see ka töötab. C++ puhul esineb sageli olukordi, kus kompileeruv programm ei tööta ning arendaja peab silumise teel viga otsima hakkama.

5.5 Arenduskogemuse võrdluse kokkuvõte

Erinevused arenduskogemuses on toodud välja tabelis 4. Kokkuvõttes oli autori kogemusel jõudlustestide kirjutamiseks Rusti ja Rusti iteraatorite kasutamine mugavam, kui C++-i ja

Streams teegi kasutamine. See on peamiselt tänu Rusti modernsetele keelemugavustele, kompilaatori rangusele ja selgematele veateadetele ning Rusti iteraatorite laialdasemale kasutusele võrreldes Streams teegiga.

Tabel 4. Arenduskogemuse võrdlus lahenduste vahel.

Võrdluspunkt	**C++ ja Streams teek**	**Rust ja Rusti iteraatorid**
Kompilaatori veateated	Mahukad ja keerulised, mõnikord eksitavad	Lühikesed, konkreetsed ja loetavad
Trafarettkood	Rohkem	Vähem
Teekide võrdlus	Üldiselt mugav, aga puudub dokumentatsioon ja laialdane kasutus	Mugav, dokumenteeritud, laialdaselt kasutatud, lihtsasti laiendatav lisateekidega
Õppimisprotsessi keerukus	Keeruline	Keeruline, aga lihtsam kui C++ puhul

6 Kokkuvõte

Lõputöö eesmärk oli uurida, kas Intel TDX-i baasil loodud lahendus, mis kasutab Rusti ja Rusti iteraatoreid on elulises andmetöötlusolukorras sobiv alternatiiv Intel SGX-i lahendusele, mis kasutab C++-i, Sharemind HI-d ja Streams teeki. Töö keskendus kahele peamisele võrdluskriteeriumile: jõudlus ja arenduskogemus.

Võrdlus teostati läbi praktilise eksperimendi, mille käigus töötas autor välja identse loogikaga jõudlustestid mõlema lahenduse jaoks ning mõõtis andmete töötlemise ajakulu. Eksperimentaalsed tulemused näitasid, et Intel TDX-il põhinev lahendus on Intel SGX-i alternatiivina igati sobiv. Siin on vastused sissejuhatuses välja toodud uurimisküsimustele:

Alaküsimus 1: Kas andmetöötluse teostamiseks vajalik funktsionaalsus on olemas?

> Intel TDX lahendus koos Rusti iteraatorite ja Itertools teegiga võimaldas rakendada kõik andmetöötluseks vajalikud funktsioonid, kusjuures saadaval olevate abimeetodite hulk oli märgatavalt suurem (vt peatükid 2.2.1 ja 2.3.1).

Alaküsimus 2: Kuidas erineb jõudlus?

> Intel TDX-i lahenduse jõudlus oli 15% kuni 35% parem (vt peatükk 4.1).

Alaküsimus 3: Kuidas erineb arenduskogemus kasutatavuse ja laiendatavuse seisukohalt?

> Arenduskogemus oli Intel TDX-i lahenduse puhul mugavam peamiselt tänu Rusti modernsetele keelemugavustele, kompilaatori rangusele ja selgematele veateadetele. Lisaks on Rusti iteraatorid laialdasemalt kasutatud ning lihtsamalt laiendatavad (vt peatükk 5).

Tulemused näitavad, et Intel TDX võib teatud juhtudel kujuneda eelistatud platvormiks konfidentsiaalsete andmetöötlusülesannete teostamisel, pakkudes paremat arenduskogemust ja jõudlust. Siiski on Intel TDX-i tarkvaraline ökosüsteem veel varajases arengufaasis, mistõttu puudub praegu laialdane tugi paljude kasutusjuhtude katmiseks. Intel SGX seevastu on küpsem, pakkudes lisaks ka väiksemat usaldatava töötluse baasi (TCB). Seetõttu võib Intel SGX olla paljudes olukordades endiselt eelistatud valik, eriti seal, kus turvalisus on tähtsam kui kiirus.

Samuti ilmnes Intel TDX-i jõudlustesti implementeerimise käigus mitmeid praktilisi väljakutseid, mille lahendamine aitas paremini mõista, kuidas sarnaseid andmeanalüüsiülesandeid Intel TDX-i kontekstis tõhusalt realiseerida.

Kasutatud kirjandus

[1] Intel Corporation. *Intel® Software Guard Extensions*. [Vaadatud: 19-12-2024]. URL: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[2] Intel Corporation. *Intel® Trust Domain Extensions (Intel® TDX)*. [Vaadatud: 19-12-2024]. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html.

[3] Cybernetica AS. *Sharemind HI White Paper*. [Vaadatud: 20-12-2024]. Jaanuar 2021. URL: https://cyber.ee/uploads/sharemind_hi_white_paper_ec24e8189a.pdf.

[4] Cybernetica AS. *Sharemind HI As Data Analysis Platform*. [Vaadatud: 20-12-2024]. URL: https://docs.sharemind.cyber.ee/sharemind-hi/4/architectural-patterns/data-analysis-platform.html#_the_streams_h_library.

[5] M. Sabt, M. Achemlal, and A. Bouabdallah. *Trusted Execution Environment: What It is, and What It is Not*. [Vaadatud: 19-12-2024]. 2015. DOI: https://doi.org/10.1109/Trustcom.2015.357.

[6] Google LLC. *Android Security Paper 2024*. [Vaadatud: 19-12-2024]. 2024. URL: https://services.google.com/fh/files/misc/android-security-paper-2024.pdf.

[7] Apple Inc. *Payment authorization with Apple Pay*. [Vaadatud: 19-12-2024]. Veebruar 2021. URL: https://support.apple.com/guide/security/payment-authorization-with-apple-pay-secc1f57e189/web.

[8] A. Paju et al. *SoK: A Systematic Review of TEE Usage for Developing Trusted Applications*. [Vaadatud: 19-12-2024]. DOI: https://doi.org/10.48550/arXiv.2306.15025.

[9] J. Young and S. Lugani. *Announcing new Confidential Computing updates for even more hardware security options*. [Vaadatud: 08-01-2025]. Oktoober 2024. URL: https://cloud.google.com/blog/products/identity-security/new-confidential-computing-updates-for-more-hardware-security-options.

[10] K. Hande. *General Availability: Azure confidential VMs with NVIDIA H100 Tensor Core GPUs*. [Vaadatud: 08-01-2025]. Sept. 2024. URL: https://techcommunity.microsoft.com/blog/azureconfidentialcomputingblog/general-availability-azure-confidential-vms-with-nvidia-h100-tensor-core-gpus/4242644.

[11] Intel Corporation. *Intel® SGX Data Center Attestation Primitives (Intel® SGX DCAP)*. [Vaadatud: 19-12-2024]. 2019. URL: https://www.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-dcap-ecdsa-orientation.pdf.

[12] I. Anati et al. *Innovative Technology for CPU Based Attestation and Sealing*. [Vaadatud: 19-12-2024]. 2013. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/hasp-2013-innovative-technology-for-attestation-and-sealing-413939.pdf.

[13] J. Randmets and A. Kisand. *An Overview of Vulnerabilities and Mitigations of Intel SGX Applications*. [Vaadatud: 19-12-2024]. 2024. URL: https://cyber.ee/uploads/D_2_116_An_Overview_of_Vulnerabilities_and_Mitigations_of_Intel_SGX_Applications_c1282b1505.pdf.

[14] Gramine Contributors. *Gramine Library OS with Intel SGX Support*. [Vaadatud: 08-01-2025]. URL: https://github.com/gramineproject/gramine.

[15] Y. Michalevsky. *Meltdown and Spectre and what it means for Intel SGX*. [Vaadatud: 08-01-2025]. Apr. 2019. URL: https://medium.com/anjuna-security/meltdown-spectre-and-what-it-means-for-intel-sgx-492ec9c8b689.

[16] Canary Bit AB. *Intel SGX vs TDX: what is the difference?* [Vaadatud: 19-12-2024]. Juuli 2022. URL: https://www.canarybit.eu/intel-sgx-vs-tdx-what-is-the-difference/.

[17] N. Weichbrodt et al. *AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves*. [Vaadatud: 17-04-2025]. Sept. 2016. DOI: https://doi.org/10.1007/978-3-319-45744-4_22.

[18] S. Verdi. *Why Rust is the most admired language among developers*. [Vaadatud: 27-12-2024]. Aug. 2023. URL: https://github.blog/developer-skills/programming-languages-and-frameworks/why-rust-is-the-most-admired-language-among-developers/.

[19] The Rust Project Contributors. *Module iter*. [Vaadatud: 27-12-2024]. URL: https://doc.rust-lang.org/std/iter/index.html.

[20] S. Klabnik and C. Nichols. *The Rust Programming Language*. [Vaadatud: 27-12-2024]. Veebruar 2023. Chap. Processing a Series of Items with Iterators. URL: https://doc.rust-lang.org/book/ch13-02-iterators.html#processing-a-series-of-items-with-iterators.

[21] S. Klabnik and C. Nichols. *The Rust Programming Language (First Edition)*. [Vaadatud: 27-12-2024]. Veebruar 2023. Chap. Iterators. URL: https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/iterators.html.

[22] *Crate itertools*. [Vaadatud: 27-12-2024]. URL: https://docs.rs/itertools/latest/itertools/index.html.

[23] Eurostat and Cybernetica AS. *Eurostat-Cybernetica project*. [Vaadatud: 27-12-2024]. 2021. URL: http://web.archive.org/web/20221109170917/https://ec.europa.eu/eurostat/cros/content/eurostat-cybernetica-project_en.

[24] F. Ricciato et al. *A proof-of-concept solution for the secure private processing of longitudinal Mobile Network Operator data in support of official statistics*. [Vaadatud: 27-12-2024]. 2021. URL: http://web.archive.org/web/20221114020902/https://ec.europa.eu/eurostat/cros/sites/default/files/unece2021_paper_eurostat_cybernetica_v6_0.pdf.

[25] Eurostat and Cybernetica AS. *mnodata-tee-poc*. [Vaadatud: 30-12-2024]. URL: https://github.com/eurostat/mnodata-tee-poc.

[26] H. Eerikson et al. *ESTAT 2019.0232 Solution Analysis*. [Vaadatud: 27-12-2024]. 2021. URL: http://web.archive.org/web/20221114020917/https://ec.europa.eu/eurostat/cros/sites/default/files/estat_2019.0232_solution_analysis_0.pdf.

[27] V. Sokk. *[ESTAT 2019.0232] Synthetic Test Data Generation*. [Vaadatud: 30-12-2024]. 2021. URL: http://web.archive.org/web/20221114020959/https://ec.europa.eu/eurostat/cros/sites/default/files/estat_2019.0232_synthetic_test_data_generation_1.pdf.

[28] D. Luu. *What's new in CPUs since the 80s?* [Vaadatud: 21-04-2025]. Jaanuar 2015. URL: https://danluu.com/new-cpu-features/.

[29] D. A. McGrew and J. Viega. *The Galois/Counter Mode of Operation (GCM)*. [Vaadatud: 10-03-2025]. URL: https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf.

[30] H. Lipmaa, P. Rogaway, and D. Wagner. *Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption*. [Vaadatud: 10-03-2025]. URL: https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/ctr/ctr-spec.pdf.

[31] NixOS Foundation. *Ad hoc shell environments*. [Vaadatud: 03-01-2025]. URL: https://nix.dev/tutorials/first-steps/ad-hoc-shell-environments.

[32] NixOS Foundation. *Declarative shell environments with shell.nix*. [Vaadatud: 03-01-2025]. URL: https://nix.dev/tutorials/first-steps/declarative-shell.

[33] Intel Corporation. *Intel® Processors Supporting Intel® SGX*. [Vaadatud: 06-01-2025]. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions-processors.html.

[34] Intel Corporation. *Overview of Load Value Injection*. [Vaadatud: 29-04-2025]. Aug. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/load-value-injection.html.

[35] *Crate bytemuck*. [Vaadatud: 06-01-2025]. URL: https://docs.rs/bytemuck/latest/bytemuck/.

[36] Intel Corporation. *What Intel® Xeon Processors Support for Intel® Trust Domain Extensions (Intel® TDX)?* [Vaadatud: 06-01-2025]. URL: https://www.intel.com/content/www/us/en/support/articles/000091103/processors/intel-xeon-processors.html.

[37] Intel Corporation. *Instruction to set up TDX host and guest*. [Vaadatud: 06-01-2025]. URL: https://github.com/intel/tdx-linux/wiki/Instruction-to-set-up-TDX-host-and-guest.

[38] Intel Corporation. *Intel® TDX Virtual Firmware Design Guide*. [Vaadatud: 06-01-2025]. Detsember 2023. URL: https://cdrdv2-public.intel.com/733585/tdx-virtual-firmware-design-guide-rev-004-20231206.pdf.

[39] The QEMU Project Developers. *About QEMU*. [Vaadatud: 06-01-2025]. URL: https://www.qemu.org/docs/master/about/index.html.

[40] Free Software Foundation Inc. *Intel® Trust Domain Extensions (TDX) on Ubuntu*. [Vaadatud: 09-04-2025]. URL: https://github.com/canonical/tdx/tree/3.2.

[41] NixOS contributors. *How Nix Works*. [Vaadatud: 11-03-2025]. URL: https://nixos.org/guides/how-nix-works/.

[42] *Documentation/9p*. [Vaadatud: 11-04-2025]. URL: https://wiki.qemu.org/Documentation/9p.

[43] SUSE LLC and contributors. *Managing systemd Services*. [Vaadatud: 15-01-2025]. Sept. 2024. URL: https://documentation.suse.com/smart/systems-management/html/systemd-management/index.html.

[44] L. Coppolino et al. *An experimental evaluation of TEE technology: Benchmarking transparent approaches based on SGX, SEV, and TDX*. [Vaadatud: 16-04-2025]. Aug. 2024. DOI: https://doi.org/10.1016/j.cose.2025.104457.

[45] *Module iter*. [Vaadatud: 16-04-2025]. URL: https://docs.rs/rayon/latest/rayon/iter/index.html.

[46] Javatpoint. *DIFFERENT COMPILERS FOR C++*. [Vaadatud: 08-01-2025]. URL: https://www.javatpoint.com/different-compilers-for-cpp.

[47] Free Software Foundation Inc. *GCC, the GNU Compiler Collection*. [Vaadatud: 29-04-2025]. URL: https://gcc.gnu.org/.

[48] Intel Corporation. *Intel® Software Guard Extensions SDK for Linux OS Release Notes*. [Vaadatud: 08-04-2025]. Sept. 2024. URL: https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_SDK_Release_Notes_Linux_2.25_Open_Source.pdf.

[49] Microsoft. *Compiling a C/C++ project*. [Vaadatud: 29-04-2025]. 2022. URL: https://learn.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=msvc-170.

[50] *Clang: a C language family frontend for LLVM*. [Vaadatud: 29-04-2025]. URL: https://clang.llvm.org/.

[51] A. Balaam. *C++ is an expert language*. [Vaadatud: 09-01-2025]. Apr. 2008. URL: https://artificialworlds.net/blog/2008/04/07/c-is-an-expert-language/.

[52] E. Zaretskii. *What does Ïnternal compiler error mean?* [Vaadatud: 08-01-2025]. URL: https://www.delorie.com/djgpp/v2faq/faq6_6.html.

[53] *Learning Rust: The Compiler is your Friend*. [Vaadatud: 21-04-2025]. Aug. 2020. URL: https://ferrous-systems.com/blog/the-compiler-is-your-friend.

[54] *Concepts library (since C++20)*. [Vaadatud: 30-04-2025]. Sept. 2024. URL: https://en.cppreference.com/w/cpp/concepts.

[55] Microsoft. *Header files (C++)*. [Vaadatud: 09-01-2025]. Aug. 2021. URL: https://learn.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=msvc-170.

[56] Microsoft. *Overview of modules in C++*. [Vaadatud: 30-04-2025]. Apr. 2025. URL: https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170.

[57] XAMPPRocky and contributors. *Tokei*. [Vaadatud: 07-03-2025]. URL: https://github.com/XAMPPRocky/tokei.

[58] M. Kline. *std::visit is everything wrong with modern C++*. [Vaadatud: 07-03-2025]. Sept. 2017. URL: https://bitbashing.io/std-visit.html.

[59] S. Klabnik and C. Nichols. *The Rust Programming Language*. [Vaadatud: 09-01-2025]. Veebruar 2023. Chap. References and Borrowing. URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[60] S. Klabnik and C. Nichols. *The Rust Programming Language*. [Vaadatud: 10-01-2025]. Veebruar 2023. Chap. Lifetimes. URL: https://doc.rust-lang.org/rust-by-example/scope/lifetime.html.

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

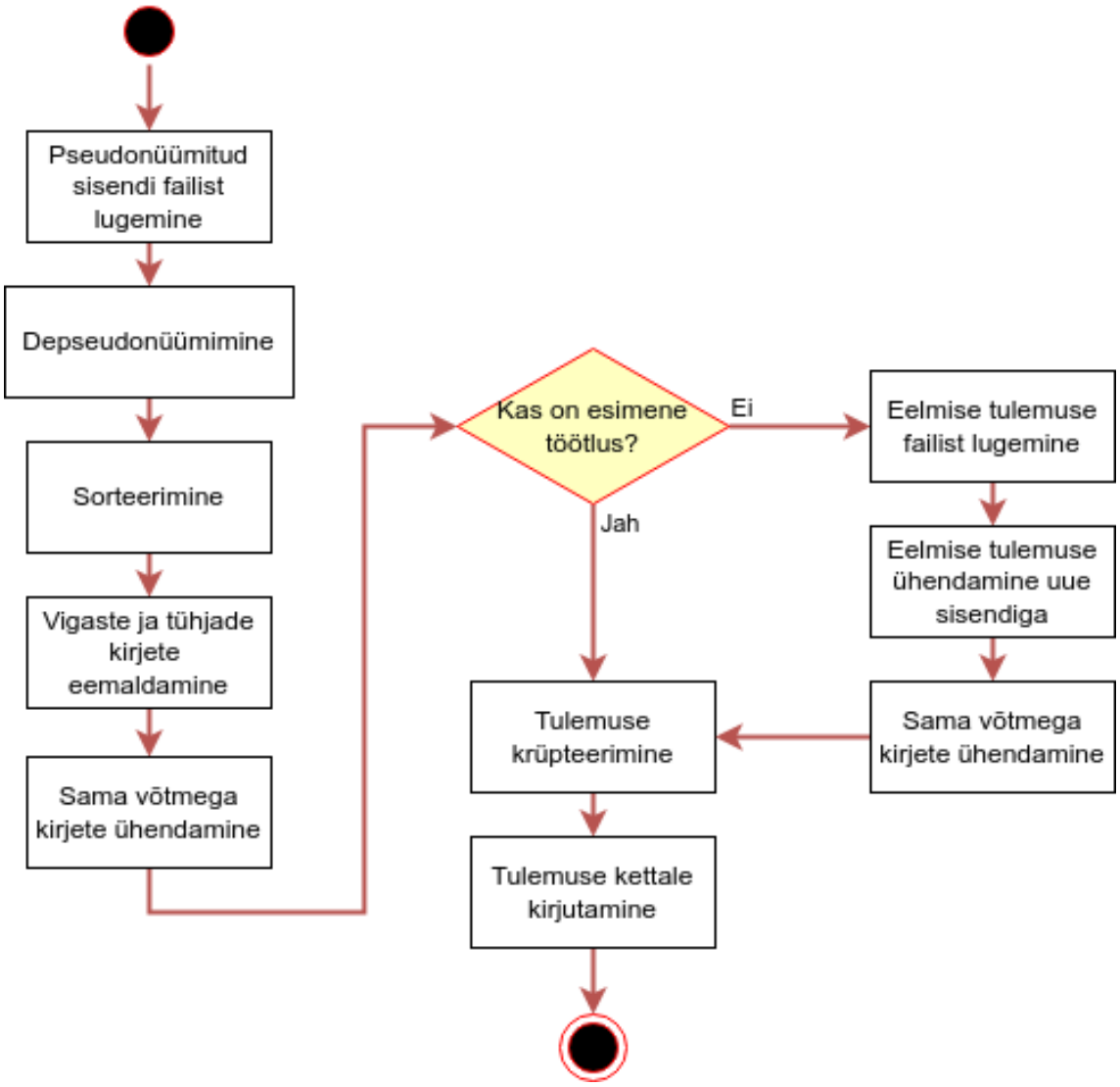
Mina, Paul Jürgens

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Spetsialiseeritud C++ andmevootöötluse teegi kasutamine Intel SGX-i ja Rusti iteraatorite kasutamine Intel TDX-i keskkonnas: võrdlev analüüs", mille juhendaja on Gert Kanter and Armin Daniel Kisand
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

12.05.2025

[1] Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Jõudlustestide loogikat kirjeldav UML skeem



Lisa 3 – GitLabi koodihoidla jõudlustestide ja skriptidega

Kõik lõputöös kasutatud jõudlustestid, skriptid ja konfiguratsioonifailid on kättesaadavad järgmises GitLabi koodihoidlas:

https://gitlab.cs.ttu.ee/pajurg/iaib

Märkus: koodihoidla on kättesaadav vaid TalTechi GitLabi sisselogitud isikutele.

Lisa 4 – Nixi kesta konfiguratisoonifail

```
{ pkgs ? import <nixpkgs> {} }:

let
  # Allows using specific Python versions in Nix.
  # https://github.com/cachix/nixpkgs-python
  nixpkgsPython = import  (fetchTarball
  ↪  "https://github.com/cachix/nixpkgs-python/archive/refs/heads/main.zip");

  # Version 3.8 is compatible with the data generator logic.
  python = nixpkgsPython.packages.x86_64-linux."3.8";

in
  pkgs.mkShell {
    buildInputs = [
      # Dependencies for SGX CPP benchmark.
      pkgs.gcc

      # Dependencies for TDX Rust benchmark.
      pkgs.rustc
      pkgs.cargo
      pkgs.virtiofsd
      pkgs.openssl
      pkgs.pkg-config
      pkgs.go

      # Dependencies for data generation.
      python
      python.pkgs.setuptools
      python.pkgs.wheel
    ];

    shellHook = ''
      source ./config.local
      echo "Loaded configuration from config.local"

      if [ -z "$PATH_TO_DATA_GENERATOR" ]; then
        echo "Error: You must specify the path to the ESTAT data generator folder by
        ↪  setting the PATH_TO_DATA_GENERATOR variable in the config.local file."
        echo "The folder can be found in the following repository:
        ↪  https://github.com/eurostat/mnodata-tee-poc"
        exit 1
      fi

      # The following is necessary for installing Python dependencies for the data
      ↪  generator.
      export DATA_GENERATOR_LD_LIBRARY_PATH="${pkgs.gcc.cc.lib}/lib"

      # Create directory for generated files.
      mkdir generated
      cd generated

      # Create a Python venv with all required dependencies for data generation. Before
      ↪  attempting to generate data, the user
```

```
    # should source this venv by using `source venv/bin/activate`. After the task is
    ↪  done, you can exit using `deactivate`.
    if [ ! -d "venv" ]; then
      python -m venv venv
    fi
    source venv/bin/activate
    pip install -r $PATH_TO_DATA_GENERATOR/requirements.txt
    deactivate

    # Fetch and create required files needed for data generation.
    cp -r "$PATH_TO_DATA_GENERATOR"/input .
    mkdir -p ./output

    cd ..

    chmod +x run-benchmark.sh
    chmod +x tdx-rust-benchmark/run-bin.sh
    chmod +x sgx-cpp-benchmark/run-bin.sh
  '';
}
```

Lisa 5 – Intel SGX jõudlustesti põhiloogika

```
void run(TaskInputs const & inputs, TaskOutputs & outputs) {
    SUPER_DUPER_TIME_GUARD("Enclave run.");

    auto const h_file_path = *inputs.argument("h_file_path");
    constexpr auto one_MiB_buffer = mebibytes(1);
    auto h_file = HFileSource(h_file_path.c_str(), one_MiB_buffer,
    ↪  sharemind_hi::StoreType::TEMPORARY_STORE);

    auto const pseudonymisation_key_file_path =
    ↪  *inputs.argument("pseudonymisation_key_path");
    uint8_t key[pseudonymisation_key_byte_length] = {};
    read_pseudonymisation_key(pseudonymisation_key_file_path, key);
    PseudonymisationKeyRef pseudonymisation_key = key;

    struct LastSeen {
        PseudonymisedUserIdentifier pseud_id;
        UserIdentifier id;

        LastSeen(HFileSource & h_file, PseudonymisationKeyRef pseudonymisation_key)
        {
            PseudonymisedUserFootprint first;
            if (h_file.peek(first)) {
                pseud_id = first.id;
                id = decrypt_pseudonym(pseudonymisation_key, pseud_id);
            } else {
                // The H file is empty, so no records will be decrypted.
                // The members can stay uninitialized.
            }
        }
    } last_seen = {h_file, pseudonymisation_key};

    auto sorted_h_file = std::move(h_file)
        >>=
            smap([&](PseudonymisedUserFootprint const & e) {
                PseudonymisedUserIdentifier pseud_id = e.id;
                if (pseud_id != last_seen.pseud_id) {
                    last_seen.pseud_id = pseud_id;
                    last_seen.id = decrypt_pseudonym(pseudonymisation_key, pseud_id);
                }
                return IdentifiedUserFootprint{{last_seen.id, e.tile}, e.i_column};
            })
        >>= sort(CMP_LAMBDA(<, IdentifiedUserFootprint, e.key), detail::mebibytes(64));

    auto cleaned_deduped_sorted_h_file = std::move(sorted_h_file)
        >>= filter([](IdentifiedUserFootprint const & e) noexcept {
                bool positive_found = false;
                for (auto const value : e.i_column) {
                    if (!std::isfinite(value)) { return false; }
                    if (value < 0) { return false; }
                    if (value > 0) { positive_found = true; }
                }
                return positive_found;
            })
```

```
    // Not using `squash` here, as in theory only one record per tile
    // per user should be in the input data.
    >>= groupBy(CMP_LAMBDA(==, IdentifiedUserFootprint, e.key))
    >>= flatMap(
            [](std::vector<IdentifiedUserFootprint> const & v,
            ↪  std::vector<IdentifiedUserFootprint> & result_vec) -> void {
                // Note: if the input H would be sanitized, `v` would
                // always only contain a single element.

                assert(!v.empty());
                result_vec.push_back(v.front());
                if (v.size() == 1) { return; }

                IdentifiedUserFootprint & result = result_vec.front();
                for (auto const & e : v) {
                    for (std::size_t i = 0;
                            i < result.i_column.size();
                            ++i) {
                        auto const a = result.i_column[i];
                        auto const b = e.i_column[i];
                        result.i_column[i] = std::max(a, b);
                    }
                }
                return;
            });

// At this point, H values are sorted by (ID, tile_index).
// Each (ID, tile_index) is unique.
// There is the invariant that the same holds for S, since the result of
// the following merge function is also sorted by (ID, tile_index) with
// (ID, tile_index) being unique.

auto const instance_id_str = *inputs.argument("instance_id");
int instance_id = std::stoi(instance_id_str);
auto const s_file_out_path = *inputs.argument("s_file_out_path");
EncStreamBase::KeyType new_s_file_key = crypto_keys[instance_id % 30];

// No outer join on first H file analysis.
if (instance_id == 0) {
    EncryptedDataSink<IdentifiedUserFootprint> sink(new_s_file_key,
    ↪  s_file_out_path);
    sink.writeFromStream(cleaned_deduped_sorted_h_file);
    return;
}

EncStreamBase::KeyType last_s_file_key = crypto_keys[(instance_id - 1) % 30];
auto const s_file_in_path = *inputs.argument("s_file_in_path");
auto s_file_in = EncryptedDataSource<IdentifiedUserFootprint>(last_s_file_key,
↪  s_file_in_path);

auto updated_s = outerJoin(
        std::move(cleaned_deduped_sorted_h_file),
        std::move(s_file_in),
        [](IdentifiedUserFootprint const & e) /* value */ { return e.key; },
        [](IdentifiedUserFootprint const & e) /* value */ { return e.key; })
    >>= smap([](std::pair<std::vector<IdentifiedUserFootprint>,
    ↪  std::vector<IdentifiedUserFootprint>> const & vv) noexcept
             -> IdentifiedUserFootprint {
            IdentifiedUserFootprint result;
            assert(vv.second.size() <= 1);
```

```
                assert(vv.first.size() <= 1);
                if (vv.first.empty()) {
                    result = vv.second.front();
                } else if (vv.second.empty()) {
                    result = IdentifiedUserFootprint{vv.first.front().key,
                    ↪  vv.first.front().i_column};
                } else {
                    result = vv.second.front();
                    for (std::size_t i = 0; i < result.i_column.size(); ++i) {
                        result.i_column[i] += vv.first.front().i_column[i];
                    }
                }
                return result;
            });

    EncryptedDataSink<IdentifiedUserFootprint> sink(new_s_file_key, s_file_out_path);
    sink.writeFromStream(updated_s);
}
```

Lisa 6 – Intel TDX jõudlustesti põhiloogika

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let start_time = Instant::now();
    println!("Starting TDX Rust benchmark.");

    let tcp_file_server_address =
    ↪  env::var("TCP_FILE_SERVER_ADDRESS").expect("\"TCP_FILE_SERVER_ADDRESS\" env
    ↪  variable must be specified");

    let h_file_name =
        env::var("H_FILE_NAME").expect("\"H_FILE_NAME\" env variable must be
        ↪  specified");

    let mut s_file_name = env::var("S_FILE_NAME").unwrap_or("/sfile".into());

    // The instance id is a value that gets incremented with each H file in a benchmark
    ↪  and basically
    // describes which day is being processed.
    let instance_id = env::var("INSTANCE_ID")
        .expect("\"INSTANCE_ID\" env variable must be specified")
        .parse::<usize>()
        .expect("\"INSTANCE_ID\" env variable must be a usize");

    // The periodic key is generated by the ESTAT data generator and is used to
    ↪  depseudonymise the input data.
    let path_to_periodic_key = env::var("PATH_TO_PERIODIC_KEY")
        .expect("\"PATH_TO_PERIODIC_KEY\" env variable must be specified");

    println!("Reading input data (H) from {}", &h_file_name);
    let pseudonymised_h_file =
    ↪  read_user_footprint_binary_file::<PseudonymisedUserFootprint>(
        &h_file_name,
        &tcp_file_server_address,
        io::DataType::Plaintext,
    )?;

    let periodic_key = read_periodic_key(&path_to_periodic_key);
    let (mut last_seen_pseud_id, mut last_seen_id); // Last seen IDs

    let mut pseudonymised_h_file = pseudonymised_h_file.peekable();
    if let Some(first) = pseudonymised_h_file.peek() {
        last_seen_pseud_id = first.key.id;
        last_seen_id = decrypt_pseudonym(&last_seen_pseud_id, &periodic_key)?;
    } else {
        return Err(Box::from("Input is empty. Stopping process."));
    }

    // Depseudonymise data and filter out entries that encounter an error when
    ↪  decrypting.
    let h_file = pseudonymised_h_file.filter_map(
        |item: PseudonymisedUserFootprint| -> Option<IdentifiedUserFootprint> {
            if item.key.id != last_seen_pseud_id {
                last_seen_pseud_id = item.key.id;
                last_seen_id = decrypt_pseudonym(&last_seen_pseud_id,
                ↪  &periodic_key).ok()?;
```

```
            }
            Some(IdentifiedUserFootprint {
                key: IdentifiedFootprintKey {
                    id: last_seen_id,
                    tile: item.key.tile,
                },
                subperiod_values: item.subperiod_values,
            })
        },
);

// Sort input data (H) by key in ascending order and remove faulty/empty data. Also
↪  get rid
// of duplicate entries from input data (H). Done by taking the maximum value from
↪  each
// subperiod.
let sorted_cleaned_deduped_h_file = h_file
    .sorted_unstable_by(|a, b| a.key.cmp(&b.key))
    .filter(|item| {
        let positive_value_found = item.subperiod_values.iter().any(|&value| value
        ↪  > 0.0);
        let contains_negative = item.subperiod_values.iter().any(|&value| value <
        ↪  0.0);
        positive_value_found && !contains_negative
    })
    .coalesce(|mut a, b| {
        if a.key == b.key {
            for i in 0..SUBPERIODS_IN_DATA {
                a.subperiod_values[i] =
                ↪  a.subperiod_values[i].max(b.subperiod_values[i]);
            }
            Ok(a) // Merge b into a
        } else {
            Err((a, b)) // Keep both
        }
    });

// At this point, H values are sorted by (ID, tile_index).
// Each (ID, tile_index) is unique.
// There is the invariant that the same holds for S, since the result of the
↪  following merge
// function is also sorted by (ID, tile_index) with (ID, tile_index) being unique.

let s_file_result =
    read_user_footprint_binary_file(&s_file_name, &tcp_file_server_address,
    ↪  io::DataType::Encrypted { instance_id });

s_file_name.push_str("-new");
match s_file_result {
    Ok(s_file) => {
        println!("Merging H and S using outer join.");
        let accumulated_s = join_h_and_s_iterators(sorted_cleaned_deduped_h_file,
        ↪  s_file);
        write_s_file(&tcp_file_server_address, &s_file_name, accumulated_s,
        ↪  &instance_id)?;
    }
    Err(e) => {
        println!(
            "Error when reading S file. Writing H to S without joining. Error: {}",
            e
```

```
            );
            write_s_file(&tcp_file_server_address, &s_file_name,
            ↪  sorted_cleaned_deduped_h_file, &instance_id)?;
        }
    };

    println!("Task successful. Result written to S file.");

    let elapsed_time = start_time.elapsed();
    println!("Time taken to process: {} ms", elapsed_time.as_millis());

    Ok(())
}
```