CHALMERS

Sandboxing

En fallstudie om sandboxlösningar för ARM-baserade embedded-system

Examensarbete inom Data- och Informationsteknik

HAMPUS HESSEL
ALBIN RYDBERG

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2018

Sandboxing

En fallstudie om sandboxlösningar
för ARM-baserade embedded-system

HAMPUS HESSEL
ALBIN RYDBERG

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg 2018

Sandboxing
En fallstudie om sandboxlösningar för ARM-baserade embedded-system

HAMPUS HESSEL
ALBIN RYDBERG

© HAMPUS HESSEL, ALBIN RYDBERG, 2018.

Handledare: Alexandra Angerd, Institutionen för Data- och Informationsteknik
Examinator: Jonas Almström Duregård, Institutionen för Data- och Informationsteknik

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet
412 96 Göteborg
Telefon: 031-772 1000

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Omslag: Bild från OpenClipart.org (Public Domain).

Institutionen för Data- och Informationsteknik
Göteborg 2018

Sammanfattning

Företaget Pilotfish Networks AB tillhandahåller en gateway för kollektivtrafikfordon. Kunder har uttryckt en önskan om att köra egen mjukvara på denna gateway vilket har säkerhetsimplikationer. Denna studie undersöker vilka möjligheter för körning av kundmjukvara i en säker miljö (en så kallad Sandbox) som finns på Pilotfish Vehicle Gateway, ett Linuxbaserat inbyggt system med ARM-processor och 128 MiB RAM. Studien är avgränsad till virtualseringsbaserad sandboxing samt har fokuserat på jämförelse och utvärdering av olika containeriseringstekniker och dess tillämplighet på systemet i fråga. Utvärderingen består av en redogörelse för installation, säkerhetsfunktioner och användning av de olika teknikerna samt benchmarking av dess prestanda. De tekniker som utvärderats är Firejail, LXC, LXD, Docker, rkt, runC, containerd samt KVM. Efter utvärdering rekommenderas containeriseringsverktyget LXC, främst för att det erbjuder operativsystemsvirtualisering likt en klassisk virtuell maskin vilket passar Pilotfish användningsfall. Verktyget är flexibelt och har god prestanda, erbjuder enkel konfiguration och installation samt har all säkerhetsfunktionalitet som behövs. LXC bedöms därför vara lämpligt för Pilotfish.

Nyckelord: Sandboxing, containerisering, Docker, Firejail, LXC, LXD, rkt, runC, containerd, KVM, hypervisor, virtualisering, namespaces, cgroups, capabilities, SELinux, AppArmor, Seccomp

Begreppsordlista

CRUD - Create, Read, Update, Delete; de fyra standardoperationerna på lagringsmedium.

Daemon - Ett program som arbetar utan direkt kontroll av en användare.

Fildeskriptor - En abstrakt pekare som används av processer för att identifiera en specifik I/O-resurs.

IPC - Inter-Process-Communication. Innefattar tekniker som gör det möjligt att skicka och ta emot meddelanden mellan processor och trådar.

Korskompilera - Att generera kod för en annan processorarkitektur än den kompilatorn körs på.

Overhead - Indirekta resurskostnader utöver ordinarie driftkostnader som uppkommer av administrativa eller implementationsspecifika skäl.

PID - Process IDentification. Numeriskt id som identifierar en körande process.

REST - Representational State Transfer. Industristandard för maskin-till-maskin kommunikation via webbteknologi.

Rotfilsystem - Ett rotfilsystem definieras som ett filsystem som utgår ifrån filsystemroten och på vilket alla andra filsystem monteras.

Tmpfs - En filsystemfunktion för temporär lagring av filer i RAM.

TTY - Teletype, en antingen simulerad eller verklig datorterminal för I/O.

UX - User Experience/användarupplevelse.

Innehåll

Figurer vii

Tabeller ix

1 Introduktion 1
1.1 Bakgrund & Syfte . . . 1
1.2 Mål . . . 2
1.3 Avgränsningar . . . 2

2 Teknisk bakgrund 3
2.1 Översikt över Pilotfish Vehicle Gateway . . . 3
2.2 Sandboxing . . . 4
2.3 Hypervisor-baserad virtualisering . . . 4
2.4 Containerisering . . . 5

3 Metod 13
3.1 Utvärderingsmetodik . . . 14
3.2 Benchmarkingsmetodik . . . 14

4 Genomförande & Resultat 17
4.1 Förberedelse av testsystem . . . 17
4.2 Utvärdering av sandboxingsverktyg . . . 18
4.2.1 LXC . . . 18
4.2.2 LXD . . . 21
4.2.3 Firejail . . . 22
4.2.4 runC . . . 24
4.2.5 containerd . . . 25
4.2.6 Docker . . . 25
4.2.7 rkt . . . 27
4.3 Sammanfattning . . . 29
4.4 Benchmarking . . . 30
4.5 Funktionalitetstester . . . 33

5 Diskussion 35
5.1 Dagens generation av VG210 . . . 35
5.2 Framtidens generation av VG210 . . . 36
5.3 Säkerhetsaspekter . . . 37
5.3.1 Etik & Miljöaspekter . . . 39

6 Slutsats & Rekommendation 41

Litteraturförteckning 43

A Appendix I
A.1 OCI-konfiguration I
A.2 Iptables konfiguration II
A.3 LXC-konfiguration III
A.4 Firejail-konfiguration V
A.5 Kompileringsflaggor för Linuxkärnan VI
A.6 Inittab konfiguration VIII

Figurer

2.1 VG210 3

4.1 En illustration för hur de underliggande linuxfunktionerna används via biblioteket liblxc för att skapa LXC-containrar. 18
4.2 En illustration över hur daemonen LXD förhåller sig till övrig arkitektur. 21
4.3 En illustration över Dockerarkitekturen som visar sambandet mellan Linuxfunktionerna, dockerdaemonen, containerd samt runC. 26
4.4 Primär- och sekundärminnesanvändning 30
4.5 Procentuell skillnad i nbenchresultat med nbench körandes inuti container jämfört med utan container. Geometriska medelvärden per olika testtyper. 31
4.6 Resultat diskbenchmarking samt nätverksbenchmarking 32

Tabeller

3.1 Hårdvaruöversikt, testsystem 13

4.1 Översikt över containeriseringsverktyg 29

1 Introduktion

Pilotfish Networks är ett Göteborgsbaserat företag som levererar system och applikationer som stöttar driften av kollektivtrafik. Pilotfish erbjuder bussoperatörer och trafikhuvudmän tjänster som möjliggör ökad operativ vinst, förbättrade interna processer och som ger stöd till det pågående samhälleliga miljöarbetet. Produkterna och tjänsterna är centrerade kring en gateway, Pilotfish Vehicle Gateway, stödd av användargränssnitt på både webb och i mobil.

En gateway är ett nav som sammankopplar olika enheter via olika nätverksgränssnitt och protokoll. Pilotfish Vehicle Gateway kopplar mer specifikt ihop IT-utrustning ombord på ett kollektivtrafikfordon med en backoffice-server. Detta sker via trådlös kommunikation såsom 4G och Wifi. Utrustningen kan vara allt ifrån biljettystem till en säkerhetskamera eller en GPS-enhet. Även motorparametrar övervakas.

Istället för att varje elektronisk enhet på ett fordon behöver ha en egen GPS-enhet med tillhörande antenner och ett eget modem kan en Pilotfish Vehicle Gateway användas som delar ut sin GPS-signal till de olika enheterna samt styr all kommunikation med backoffice-servrar. Detta medför vinster genom både minskad resursanvändning samt genom enklare felsökning då det är mindre komplicerat att upptäcka fel med en GPS-signal och nätverksanslutning än för flera separata.

1.1 Bakgrund & Syfte

På gatewayen körs idag huvudsakligen Pilotfish mjukvara. Samtidigt har många operatörer inom kollektivtrafiksindustrin egenutvecklad mjukvara för att styra andra system på sina fordon, till exempel utrop av busshållplatser. Det blir redundant och dyrt att som operatör införskaffa mer hårdvara då gatewayen redan är installerad och skulle vara fullt kapabel att köra kundens mjukvara. Det är dock inte problemfritt att helt släppa tyglarna fria för tredjepartsmjukvara. Det finns ett antal kritiska processer som inte får störas av krävande eller dåligt skriven mjukvara. Riskerna för sådana störningar måste minimeras; detta kan uppnås genom att tredjepartsmjukvara körs i en begränsad operativsystemmiljö, en så kallad Sandbox.

1.2 Mål

Målet med examensarbetet är att utreda vad det finns för Sandboxtekniker att tillgå och vilka tekniker som är lämpliga att använda på dagens version av fordonsgatewayen samt på en framtida version. Utredningen skall ta hänsyn till aspekter såsom CPU-användning, minnesåtgång, påverkan på Pilotfish mjukvara och användarvänlighet. Framtagandet av ytterligare bedömningskriterier om så bedöms vara lämpligt samt till vilken grad säkerhetsaspekter skall tas i beaktande följer från detta som ytterligare ett mål.

Denna utredning skall leda rekommendationer för vad som bör användas på dagens fordonsgateway och i framtiden. I rekommendationer för framtiden bör också ingå en del om hårdvaru-val/krav, t.ex. val av processor, minne etc. Test av lämplig rekommenderad teknik ske på gatewayen.

1.3 Avgränsningar

Sandboxing kan genomföras på en mängd olika sätt. Detta arbete är dock avgränsat till sandboxing som bygger på virtualisering. Faktumet att ARM-processorn på gatewayen inte stödjer hårdvaruvirtualisering samt att gatewayen har väldigt begränsade minnesresurser ledde till bedömningen att hypervisor-baserad virtualisering inte är lämplig. Arbetet avgränsades därför ytterligare till att enbart innefatta containeriseringslösningar, som till skillnad från hypervisor-baserad virtualisering kräver mindre resurser.

Då projektet är tidsbegränsat har ett begränsat antal av de sandboxingverktyg som är vanligast och mest omnämnda i den tillgängliga litteraturen valts ut. Alternativ som inte stödjer ARM eller som inte är aktivt underhållna har valts bort.

2 | Teknisk bakgrund

Detta kapitel beskriver den hårdvara projektet är inriktat mot samt ger en beskrivning av vad sandboxing är. Olika tekniker för att skapa en sandbox beskrivs, däribland containerisering. En översikt ges också över hur containerisering fungerar i praktiken.

2.1 Översikt över Pilotfish Vehicle Gateway

Figur 2.1 visar den version av fordonsgatewayen, VG210, som fanns tillgänglig under vintern 2018 vilket är den version som arbetet huvudsakligen är inriktat mot. Den övergripande hårdvarukonfiguration för denna version av VG210 presenteras nedan. Senare på våren kom även en version med 512 MiB primärminne. Denna version var dock inte tillgänglig i tid för tester så istället användes ett utvecklingskort av typen Beagle Bone Black som är hårdvarumässigt lik VG210 men med 512 MiB RAM[1].

```
Enkelkärnig ARM Cortex A8
Processor @ 800Mhz
128 MiB DDR3 RAM
4GiB eMMC flashminne
2 x RS232
1 x USB v2.0 high speed host
8+1 x Gbit RJ45 Ethernet
4 x Digitala utgångar
5 x Digitala ingångar
3 x Analoga ingångar
3 x CAN (SAE J1939, FMS)
1 x K-line (ISO9141)
1 x RS485
1 x GPS-modul
1 x GSM/UMTS/LTE-modem
1 x Wifi-gränssnitt
```

Figur 2.1: VG210

På VG210 körs en egenbyggd distribution av Linux baserad på Busybox [1]. Busybox är ett program som samlar en mängd verktyg som brukar höra till en dis-

[1] Se även kapitel 3

tribution (till exempel mkdir eller modprobe) i en enda binär. Detta möjliggör skapandet av väldigt små distributioner för inbyggda system. Distributionen på VG210 är väldigt avskalad och innehåller mycket mindre verktyg än en vanlig Linuxdistribution brukar göra. Kärnan är en egenkompilerad version av Linux 3.12.24 där den mesta av funktionaliteten som inte används har inaktiverats för att minska storleken.

2.2 Sandboxing

Sandboxing är en datasäkerhetsterm som innebär att ett program körs på ett sådant sätt att det inte kan orsaka skada på värdsystem eller delade resurser [2]. Detta kan åstadkommas på olika sätt. Ett alternativ är att programmet övervakas och stoppas. Ett exempel på en sådan lösning är antivirusprogram. Antivirusprogram hindrar i regel inte helt tillgången till resurser utan övervakar och garanterar att de inte används på ett felaktigt sätt. En annan lösning är att programmet helt separeras från värdsystemets exekveringsmiljö och placeras i en virtuell exekveringsmiljö. Detta görs med syftet kapsla in programmet och de säkerhetsproblem som kan uppkomma och således hindra dem från att påverka värdsystemet. Sandboxing kan också referera till andra metoder för att åstadkomma säkerhetslager mellan olika datatekniska företeelser. Det kan handla om systeminstruktioner som bara får exekvera under vissa förutsättningar eller om kod som bara får köras om den är signerad på ett sätt som garanterar att den kommer från en pålitlig källa.

I en sandbox kan man ofta tillämpa restriktioner på de program som körs. Restriktionerna kan till exempel begränsa tillgång till olika fysiska resurser på systemet eller kommunikation med andra processer. Detta för att undvika att sandboxade program stör värdsystemet genom använda en oproportionerligt stor mängd av de tillgängliga resurserna.

2.3 Hypervisor-baserad virtualisering

Det är möjligt att simulera all mjukvara och hårdvara som krävs för att ett fullt virtuellt system ska kunna köras men det är ineffektivt. Ett modernare alternativ är så kallad hypervisor-baserad virtualisering som bygger på översättning mellan instruktioner på gästsystemet till instruktioner som kan köras direkt på värdhårdvaran [3]. Ett program på värdsystemet, “hypervisorn”, skapar och hanterar vir-

tuella maskiner och är ansvarig för att operationer på gästsystemet översätts till operationer som kan exekveras på hårdvara. Virtuella maskiner av denna typ har hög säkerhet och flexibilitet eftersom hypervisorn har full kontroll över vad som får utföras på den fysiska hårvaran. Den huvudsakliga nackdelen är mängden overhead; att köra flera virtuella operativsystem är krävande trots hypervisorteknik.

2.4 Containerisering

Containerisering är till skillnad från annan virtualisering en metod som bygger på separation av resurser mellan virtuella miljöer inom operativsystemet så att miljöerna framstår som separata system. Detta trots att programmen i de olika miljöerna fortfarande körs på samma operativsystemkärna [4]. Sandboxmiljöer skapade med denna metod kallas för containrar.

Containerisering har möjliggjorts av framtagandet av en uppsjö olika Linuxfunktioner som tillsammans kan användas för att skapa en virtuell sandboxmiljö. Dessa Linuxfunktioner beskrivs i detalj i följande avsnitt. Eftersom all kod exekveras på samma operativsystemkärna är mängden overhead låg.

Ofta görs en indelning mellan två stora designfilosofier; containerisering på operativsystemsnivå samt containerisering på applikationsnivå. **Containerisering på operativsystemsnivå** beskriver de implementationer som använder containerisering för att uppnå något som liknar en klassisk virtuell maskin [4]. Detta åstadkoms genom att ett eget rotfilsystem inkluderas i varje container som innehåller alla de bibliotek, konfigurationsfiler och binärer som utgör en komplett operativsystemsmiljö. Filsystemet kan dock vara minimalt och det finns rotfilsystem som är så små som 8 MiB. Tekniken är vanlig om man vill köra flera väsenskilda applikationer i en och samma container eller använda flera olika Linuxdistributioner på ett och samma värdsystem.

Containerisering på applikationsnivå beskriver de implementationer som fokuserar på att containerisera enstaka applikationer för att uppnå en säker och tydlig separation mellan olika tjänster [4]. Ett helt rotfilsystem krävs i regel ändock av containeriseringsimplementationen men oftast finns funktionalitet för att automatiskt generera rotfilsystem innehållande allt som en applikation kräver. I vissa fall kringgås kravet på separat rotfilsystem genom direkt användning av delar av värdsystemets dito. Värt att notera här är att mängden overhead vid applikationscontainerisering ökar med antalet applikationer vilket det inte gör på samma sätt vid ett operativsystemsdito.

Det finns en öppen standard, **OCI Runtimespecifikationen**, med målet att standardisera hur en container konfigureras och körs [5]. Standarden specificerar ett konfigurationsformat samt hur en containers filstruktur ska se ut. En så kallad OCI-filsystembundle består av ett rotfilsystem samt en config.json fil. Det finns också specificerat hur en bundle ska paketeras för att kunna laddas ner och packas upp som en OCI-Image. Några av teknikerna som projektet avser att utvärdera följer standarden och containrar av denna typ går därmed att porta mellan dessa implementationer.

Linuxkärnan & dess säkerhetsfunktioner

Kärnan i ett operativsystem är den mjukvara som implementerar kommunikation med hårdvara och minne, process-schemaläggning och andra liknande funktioner. Linuxkärnan är en monolitisk operativsystemskärna, det vill säga en kärna som består av en enda statisk binärfil och körs som en enda process [6]. Den är högst konfigurerbar och gjord för att det ska vara enkelt att kompilera en anpassad kärna som passar det system den ska användas på. Nästan alla funktioner som det finns stöd för går att aktivera eller inaktivera med hjälp av kompileringsflaggor. Som tidigare nämnts innebär containerisering i praktiken att man använder ett antal säkerhetsfunktioner i Linuxkärnan. För att effektivt kunna använda containeriseringsimplementationer är det viktigt att förstå hur dessa funktioner fungerar. Därför följer nedan en beskrivning av samtliga dessa säkerhetsfunktioner.

Chroot

Chroot – changeroot – är en operation som byter ut filsystemroten för en process och dess subprocesser. En chroot-operation resulterar i vad som brukar kallas för ett fängelse; en exekveringsmiljö där en eller flera processer enbart har tillträde till en delmängd av värdens filsystem [7]. Detta är väldigt användbart vid containerisering då en container i regel inte bör ha tillgång till hela filsystemet.

Namespaces

För att isolera systemresurser från varandra och från processer av olika slag används Linuxfunktionen namespaces. Att använda namespaces är ett effektivt sätt att skapa restriktioner på vad processer kan se [7]. Funktionen fungerar genom att

globala resurser virtualiseras. Olika unika instanser av varje virtualiserad resurs kan sedan skapas och associeras med olika processer. För processerna som är associerade med namespacet framstår det som att de har tillgång till den faktiska resursen trots att de egentligen bara har tillgång till en virtuell instans. Ändringar i en virtuell instans syns inte i andra instanser.

Nedan följer en beskrivning av vilka olika typer av namespaces som finns och dess egenskaper.

- **IPC**. IPC-namespaces förhindrar vissa typer av interprocess-kommunikation mellan processer i olika namespaces.
- **Network**. Nätverksnamespaces kan ge olika processer olika bild av nätverksstacken och alla dess delar såsom nätverksinterfaces. Ett fysiskt nätverksgränssnitt kan bara tillhöra ett namespace åt gången.
- **Mount**. Med hjälp av mount-namespaces kan olika processer ha olika bild av filsystemets monteringspunkter.
- **PID**. PID-namespaces ger olika processer olika bild av vilka PID processer har och vilka processer som körs.
- **UTS** UTS står för Unix Time Sharing system. UTS-namespaces låter olika processer ha olika syn på systemets domän- och värdnamn.
- **User**. Med user-namespaces kan olika processer ha olika bild över användarnamn och UID:n.

Namespaces är grundstenen i containerisering då det är den funktionalitet som överhuvudtaget möjliggör att en containerprocess kan ha en annan syn på globala resurser än värdsystemet och andra processer.

CGroups

ControlGroups är en funktion som begränsar processers tillgång till och användning av olika resurser. Funktionen kan användas för att bland annat begränsa och mäta mängden CPU-tid, nätverksbandbredd och minne för en eller flera processer [7]. CGroups kan också användas för att begränsa tillgång till I/O-enheter.

En CGroup består av en grupp processer som tilldelas resurskvoter som kärnan ser till efterlevs. Det går att sätta olika prioritet på olika CGroups om de skulle

tilldelas samma resurser så att kärnan kan fördela resurserna på det sätt som avses. En CGroup får och kan inte använda mer resurser än den mängd som har tilldelats till den. CGroups kan vara mycket användbart ur ett containeriseringsperspektiv då det kan vara önskvärt att inte låta en container använda oproportionellt mycket resurser och på så sätt störa annan funktionalitet.

Rlimit

Resourcelimit är en Linuxfunktion som går att likna vid CGroups, då det är en annan metod för att begränsa resurstillgångar för en process [6]. Rlimits kan användas tillsammans med CGroups och bidra med ett ytterligare begränsningslager eller användas självständigt. Rlimits skiljer sig från CGroups i att begränsningar anges per process och inte per grupp. Det finns ett antal olika begränsningar som kan anges däribland CPU, primärminne och antal fildeskriptorer.

SELinux & AppArmor

Användning av någon av SELinux och AppArmor ändrar operativsystemets behörighetsprotokoll från DAC, vilket är standard, till MAC [8]. DAC - Discretionary Access Control - är ett åtkomstprotokoll som kännetecknas av att en fil eller resurs själv vet vilka användare som får använda den och på vilka sätt. MAC - Mandatory Access Control - är ett åtkomstprotokoll som kännetecknas av att åtkomst inte bara är beroende av behörigheter hos en resurs utan också av en global åtkomstprofil som kan konfigureras på olika sätt.

SELinux (Security Enhanced Linux) och AppArmor introducerar applikationsunika profiler. Utan SELinux eller AppArmor kontrolleras försök att använda en enskild resurs mot resursens behörighetsmatris där läs- skriv- och exekveringsbehörighet lagras. Använder man däremot något av dessa verktyg appliceras en behörighetsprofil på en process vid körning och restriktionerna utökas till att innefatta allt ifrån begränsningar på filsystem- och resurstillgång till kontroll över kapabiliteter. AppArmor och SELinux tillhandahåller i princip identisk funktionalitet men genom olika implementationer.

Vid containerisering kan SELinux eller AppArmor användas som ett ytterligare kompletterande lager med säkerhetsregler. På så sätt skapas ett ökat skydd för värdsystemet från olika säkerhetsbrister och försök att få obehörig tillgång till värdsystemet från containern.

seccomp & seccomp-bpf

Med hjälp av systemanropet seccomp() kan en process gå in i ett säkert tillstånd där endast ett väldigt begränsat antal systemanrop är tillåtna; exit(), sigreturn(), read() samt write(). Processen tillåts inte heller öppna några nya fildeskriptorer. En utbyggnad av seccomp-funktionaliteten har senare introducerats: seccomp-bpf. I och med utbyggnaden finns nu stöd för att med filter kunna begränsa exakt vilka systemanrop en process får använda och hur [7].

Seccomp är mycket användbart vid containerisering då det ger en hög grad kontroll över vad processer i en container kan göra. Funktionen kan med fördel användas för att blockera en rad systemanrop som skulle kunna göra skada på värdsystemet.

Kapabiliteter

Innan version 2.2 av Linuxkärnan fanns det inga befogenhetskontroller för program som kördes som rootanvändaren på Linux – de hade alltid möjlighet att göra allt. I moderna versioner går det att dela upp befogenheter i olika enheter, kapabiliteter, som kan aktiveras eller avaktiveras på filbasis [7]. Det finns en mängd olika kapabiliteter. Ett exempel är `CAP_SYS_TIME` som, om aktiverad, låter ett program att sätta systemklockan.

Containeriseringsverktyg kan i regel kontrollera vilka kapabiliteter en containerprocess ska ha. Detta kan användas för att öka säkerheten och kontrollera exakt vad processer i en container har för befogenheter. Kapabiliteten `CAP_SYS_ADMIN` är speciellt intressant för containerisering då den ger många befogenheter som traditionellt kräver rot-behörighet så som montering av delar av filsystemet [6]. Det kan i många fall vara önskvärt att inte ha kvar denna och andra liknande kapabiliteter i en container.

Oprivilegierade containrar

Det finns ett säkerhetsproblem med "vanliga" containrar; det finns i regel en ett-till-ett relation mellan användare i och utanför containern även om de befinner sig i olika user-namespaces. Med andra ord, användar-id 0 inuti containern är även användar-id 0 utanför containern. Skulle en process som körs som rot i containern på något sätt lyckas bryta sig ut ur den så hade den haft rot-tillgång till hela värdsystemet [9]. Opriviligerade containrar är en metod för att lösa det problemet

genom att översätta användar-id i containern till helt andra id på värdsystemet. På så sätt kan processer som körs som rotanvändaren inuti containern egentligen köras som en helt annan användare, till exempel användar-id 1000, på värden. Då användare 1000 kan vara helt opriviligerad ökar en sådan översättning säkerheten.

Följande exempel visar hur detta kan se ut i praktiken. Överst syns ett utdrag från en lista av körande processer i containern och nedan från värden. Båda utdrag visar samma process, notera att användar-id (UID) är helt olika.

```
[root@container /]# ps -ef | head -n 2
UID          PID   PPID  C STIME TTY          TIME CMD
root           1      0  0 10:22 ?        00:00:00 /sbin/init

[root@host /]# ps -ef | grep 26204 | head -n 2
UID          PID   PPID  C STIME TTY          TIME CMD
1000       26204  26200  0 12:22 ?        00:00:00 /sbin/init
```

Checkpoint/Restore in Userspace

Checkpoint/Restore in Userspace, CRIU, är ett verktyg för Linux som gör det möjligt att ta en ögonblicksbild av en process och lagra dess tillstånd. Det lagrade tillståndet kan sedan återställas [10]. Detta gör det möjligt att pausa och flytta levande containrar, vilket är användbart om man till exempel vill byta ut hårdvaran men ha kvar mjukvaran i exakt samma tillstånd eller om man vill installera en container som redan befinner sig i ett specifikt tillstånd på flera olika system. Projektet utvecklades från början för att användas vid virtualisering och flera containeriseringsverktyg har stöd för integration med CRIU.

3 Metod

Arbetet kommer delas upp i ett antal faser. Första fasen kommer bestå av undersökning av vilka sandbox-tekniker som skulle kunna vara lämpliga samt att sammanställa dessas egenskaper på ett teoretiskt plan. Målet är att i denna fas sålla bort tekniker på grunder som bristande hårdvaru- eller operativsystemstöd.

Sandboxingimplementationerna som tagits vidare i första urvalet kommer sedan att installeras på en Orange Pi Zero [11]; ett utvecklingskort vars hårdvara kommer relativt nära hårdvaran på VG210. Orange Pi:en har dock betydligt mer RAM-minne (512 MiB) samt en fyrkärnig processor. På Orange Pi:en kommer vi att köra en större Linuxdistribution: en ARM-version av Debian. Anledningen till att en Orange Pi är lämplig som första testsystem istället för att direkt testa på VG210 är att det förenklar utvärderingen ett tidigt skede; Orange Pi blir en flexiblare testplatform där själva installationen går snabbt. Test på VG210 kräver ett antal ytterligare steg så som omkompilering av Linuxkärnan. Grundantagandet är att om inte en kandidat fungerar på Orange Pi så är det väldigt osannolikt att den kommer att fungera på VG210. De implementationer som fungerar bra på Orange Pi testas sedan på VG210.

I tredje fasen skall mer genomgående benchmarking utföras för att på djupet undersöka huruvida någon implementation introducerar mer overhead än någon annan samt hur mycket denna eventuella overhead påverkar den primära funktionaliteten på VG210. Ett utvecklingskort av typen Beagle Bone Black[12] är också tillgängligt. Detta kort har hårdvara väldigt lik VG210 men mer RAM-minne (512 MiB). I de fall där containeriseringsimplementationerna inte fungerar på VG210 på grund av resursbrist kan Beagle Bone-kortet därför användas som prototyp för en framtida version av VG210 under vår testning. I denna fas kommer också olika funktionalitetstester att genomföras. Vilka dessa tester är kommer att tas fram i samråd med Pilotfish. En översikt över all tillgänglig hårdvara visas i tabell 3.1.

Tabell 3.1: Hårdvaruöversikt, testsystem

Enhet	CPU	RAM	Kärna
Orange Pi Zero	Fyrkärnig H2 Cortex-A7 1GHz	512MiB DDR3	4.14.14
VG210	Enkelkärnig Cortex-A8 1GHz	128MiB DDR3	3.12.24
BeagleBone Black	Enkelkärnig Cortex-A8 1GHz	512MiB DDR3	3.12.24

3.1 Utvärderingsmetodik

Sandboxteknikerna kommer att utvärderas kontinuerligt med hänsyn taget till följande övergripande kriterier:

- Hur mycket overhead implementationen introducerar. Även om mer rigorösa jämförelser av de olika kandidaterna ska genomföras i ett senare skede är det möjligt att tidigt ta generell overhead i beaktning. Faktorer att ta hänsyn till kan vara storleken på programmen och biblioteken som installeras samt RAM- & CPU-användning.
- Om de säkerhetsfunktioner som krävs för att uppnå projektets mål finns, går att konfigurera och ifall konfigurationen av dem är användarvänlig.

3.2 Benchmarkingsmetodik

För detaljerad benchmarking på VG210 kommer ett antal verktyg att användas vilka presenteras nedan. Benchmarkningsverktygen kommer att köras dels utan container för att få ett riktvärde samt inuti containrar som skapats av de olika sandboxingsverktygen. Samtidigt som benchmarkning körs kommer även RAM- och CPU-användning för containerprocesserna att mätas.

Nbench

För att undersöka huruvida sandboxen introducerar någon CPU- och RAM-overhead på vanliga operationer kommer verktyget nbench användas. nbench är ett verktyg som kör tio olika algoritmer enormt många gånger och räknar hur många iterationer i snitt som går att köra per sekund [13]. Algoritmerna är speciellt utvalda för att testa CPU och minneshastighet och innehåller sortering av array med långa heltal, sortering av strängar, olika bitmanipulationer, emulerade flyttalsoperationer, uträkning av fouriertransformer, en tilldelningsalgoritm, Huffmanns kompressionsalgoritm, IDEA-krypteringsalgoritmen, en neural nätverkssimulator samt lösning av linjära ekvationer.

Disk-benchmarking

För benchmarking av disk kommer verktyget dd som redan finns installerat på VG210 användas. dd är ett verktyg som huvudsakligen används för kopiering och konvertering av data [14]. Med dd kommer vi att skriva en fil full med nollor och se till att all skriven data synkroniseras till disk samtidigt som vi tar tiden på detta. Detta används som en enkel typ av diskbenchmark.

Iperf3

För att undersöka nätverksoverheaden kommer vi att använda verktyget iperf3. iperf3 är ett verktyg speciellt framtaget för att benchmarka IP-nätverk [15]. Verktyget visar maximal möjlig bandbredd mellan två enheter samt eventuellt mängden förlorade paket. Iperf består av en server och en klient och det är bandbredden i anslutningen mellan dessa som testas.

En iperf3 server kommer att sättas upp inuti containern som ska benchmarkas samt utan container. En klient kommer köras på en laptop ansluten till samma nätverk som VG210 & Beagle Bone via Gigabit-ethernet.

4 | Genomförande & Resultat

Detta kapitel presenterar resultatet av de olika tester och jämförelser som utförts. Först presenteras de sju kandidaterna och förberedelsen av testsystemet. De sju kandidaterna och dess egenskaper samt hur väl lämpade de är för projektets ändamål beskrivs sedan mer ingående. Slutligen presenteras resultaten av den benchmarking som utförts.

4.1 Förberedelse av testsystem

Förberedelsen av Orange Pi bestod av installation av operativsystemet. Eftersom kärnan på VG210 (som även används på Beagle Bone) är konfigurerad för att vara så avskalad som möjligt saknas ett antal funktioner som krävs för containerisering. Dessa aktiverades via kärnans konfigurerings-GUI. Alla funktioner som behövde aktiveras för att samtliga implementationer skulle fungera är listade i appendix 5. Kärnan korskompilerades sedan för ARM och installerades på VG210 samt BeagleBone. För kompilering användes Linaros kompilatorverktygskedja baserad på GCC 4.7. Linaro är en organisation som arbetar med att skapa öppen mjukvara för ARM [16]. Även containeriseringsverktygen behövde naturligtvis korskompileras.

Alla implementationer tillåter användaren att välja bort vissa funktioner vid kompilering. En avvägning gjordes att inte aktivera SELinux eller AppArmor samt att inte försöka skapa opriviligerade containrar. Detta för att förenkla kompilering och installation. En diskussion rörande vilka säkerhetesfunktioner som bör aktiveras vid användning i faktisk produktion återfinns i Diskussionskapitlet.

Rotfilsystemet som användes för att testa alla implementationer (förutom Firejail) skapades med verktyget Buildroot. Buildroot är ett interaktivt verktyg speciellt designat för att förenkla skapandet av korskompilerade inbyggda Linuxsystem [17]. Buildroot används sedan tidigare på Pilotfish för att bygga distributionen som används på VG210. Rotfilsystemet som skapades baserades på samma konfiguration som VG210-distributionen med följande två ändringar:

- Först tömdes konfigurationsfilen /etc/fstab som listar alla monteringspunkter som automatiskt ska monteras vid uppstart. Detta eftersom containeriseringsverktyget och inte operativsystemet ska sköta montering för containrar.

- Även inittab-konfigurationen (/etc/inittab) som styr vad som sker vid upp-

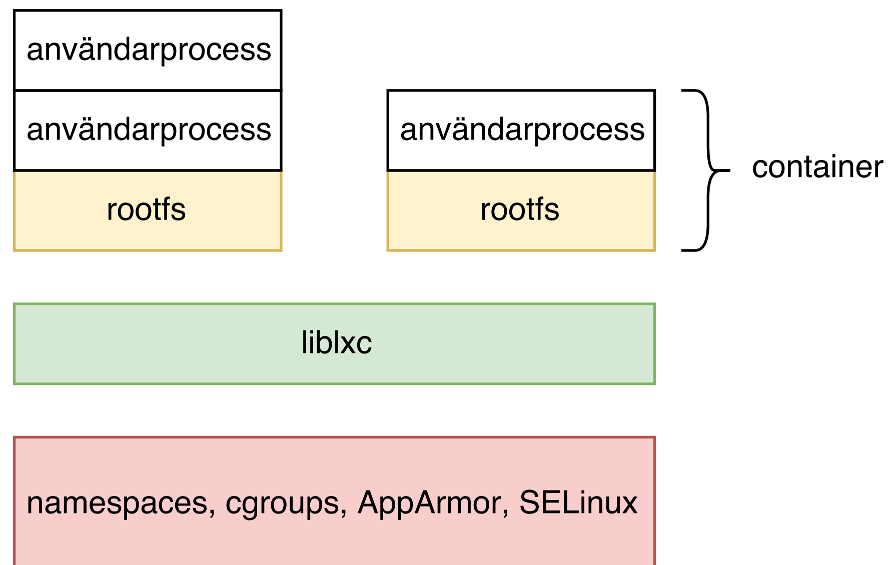
start, avstängning och omstart redigerades för att förhindra konflikter. Den ändrade inittab:en återfinns i sin helhet i appendix 6.

4.2 Utvärdering av sandboxingsverktyg

De sju verktygen som slutligen valts ut för testning är LXC, LXD, Firejail, runC, containerd, Docker samt rkt. Följande sektion innehåller en redogörelse för utvärderingen av dessa. Beskrivningarna innefattar dess egenskaper på ett mer teoretiskt plan samt hur de har byggts och testats på de olika testsystemen.

4.2.1 LXC

LXC, LinuX Containers, är en av de äldsta implementationerna av containerisering [4]. LXC består av ett antal verktyg för att skapa, övervaka och styra containrar samt ett bibliotek, liblxc, som gör LXC integrerbart i andra projekt.



Figur 4.1: En illustration för hur de underliggande linuxfunktionerna används via biblioteket liblxc för att skapa LXC-containrar.

LXC är en implementation på låg abstraktionsnivå – det är inget mer än ett gränssnitt mot funktioner som redan finns i Linuxkärnan. LXC är en implementation av

operativsystemscontainerisering men det finns inga hinder för att använda LXC för att containerisera enskilda applikationer.

LXC har stöd för att automatiskt skapa rotfilsystem och konfigurationer baserade på så kallade templates. Dessa är i regel definierade per distribution. I princip är templates bara Bash-skript som utför ett antal steg för att skapa ett rotfilsystem av önskad typ, så som att packa upp ett rotfilsystem från en tar och skapa en konfigurationsfil. Sedan LXC version 3.0.0 kan LXC användas för att köra containrar på OCI-format via en speciell OCI-template.

LXC fungerar på alla Linuxsystem där kärnan har stöd för namespaces och CGroups (version 2.6.32 eller nyare). Vill man använda fler säkerhetsfunktioner som introducerats i senare version av Linuxkärnan behöver man dock naturligtvis använda en nyare version. För att kunna köra opriviligerade containrar med LXC krävs ett verktyg som hanterar Cgroups, exempelvis cgmanager, samt stöd för uid & gid konfiguration i operativsystemets shadowfil. Dock har som tidigare nämnt inte fokus varit på att skapa opriviligerade containrar.

Det går att göra en snapshot – en ögonblickskopia – av en körande container som kan sparas och återställas senare. Detta kräver dock att CRIU-verktygen finns installerade. CRIU-funktionaliteten har ej testats.

Installation och konfiguration

LXC testades först på Orange Pi och testningen inleddes med installation av de binärfiler och bibliotek som LXC består av. När detta var gjort behövde ett rotfilsystem skapas. I detta skede användes den inbyggda funktionaliteten för att skapa ett rotfilsystem och BusyBox-templatet.

För att testa nätverkskonfiguration containeriserades en Apache webbserver. Apache installerades manuellt i det skapade rotfilystemet.

Nätverket behövde delvis konfigureras manuellt. I konfigurationsfilen som genererats för containern kan man specificera ett virtuellt nätverksgränssnitt för containern. Gränssnittet skapas automatiskt och bryggas ihop med ett virtuellt nätverksgränssnitt som skapas på värden när containern startas. Själva bryggan måste skapas manuellt, exempelvis med kommandot brctl, och på värden måste bryggan tilldelas samma IP-address som containerns gateway för att utgående trafik från containern ska fungera. Se appendix 3 för exempel på nätverkskonfiguration.

För att containern ska kunna köra till exempel en webbserver som Apache krävs att trafik på port 80 till värden skickas via nätvärksbryggan till containern och att svarstrafik från containern skickas vidare. Detta går att åstadkomma på olika sätt, bland annat genom att använda Linuxkärnans inbyggda brandvägg iptables. Ett exempel på en iptables-konfiguration återfinns i appendix 2. Det är måhända lite omständligt att göra en iptables-konfiguration manuellt för varje container, men det hade varit enkelt att med ett skript automatisera detta vid en containers uppstart.

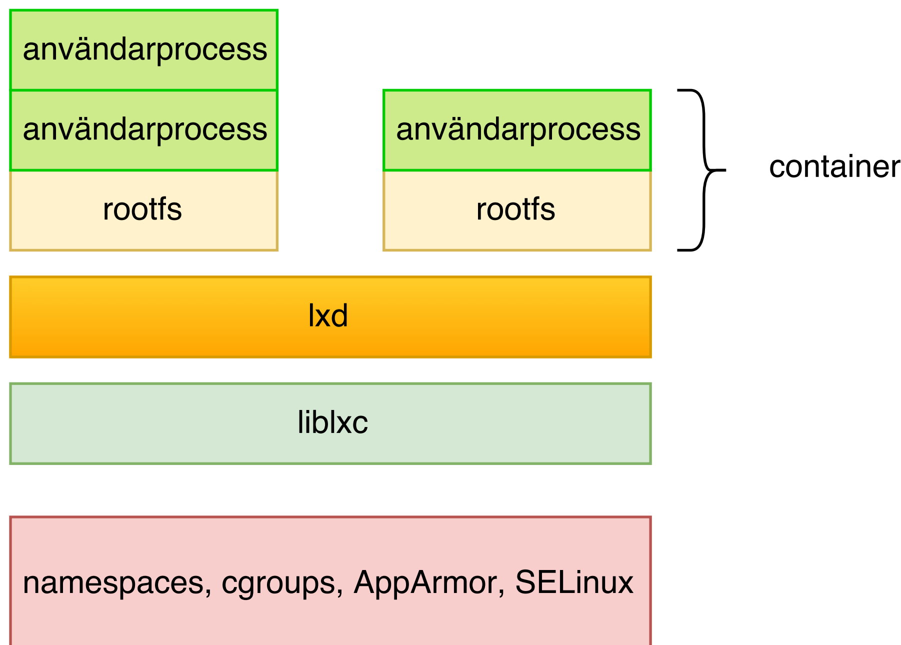
LXC testades även på VG210. Detta skedde i stort sett utan problem. En del nyare Linux-verktyg förutsätter att mappen /run existerar och är monterad som tmpfs; så även LXC. Detta måste göras antingen varje boot via mount eller via fstab. Ett annat problem är att de templates som används för att automatiskt skapa rotfilsystem ej fungerar när LXC är korskompilerat. De fungerar heller inte om inte Bash finns. Detta var dock inget större problem då ett rotfilsystem hade skapats tidigare med Buildroot och kunde placeras på flashminnet tillsammans med en konfigurationsfil. Den konfiguration vi använde vid test på VG210 återfinns i appendix 3. LXC testades kort även på Beagle bone black där det fungerade felfritt.

Önskas funktionalitet för att automatiskt skapa rotfilsystem behöver nya templates skapas. Detta är relativt trivialt; ett template är som sagt bara ett skript som kopierar ett rotfilsystem från någon källa (till exempel ett tar-arkiv) och skapar en enkel konfigurationsfil.

LXC har en lagringsstorlek på ca 10 MiB utan rotfilsystem. Konfigurationsmöjligheterna är stora och enkla att använda.

4.2.2 LXD

LXD är ett administreringsverktyg för LXC-containrar med syftet att göra hantering av LXC-containrar smidigare. Verktyget består av en daemon tillsammans med en utökad kommandoradsklient [4]. Figur 4.2 beskriver relationen mellan LXD och dess beståndsdelar. LXD har likt LXC det uttryckliga målet att användas för att skapa och hantera containrar på operativsystemsnivå.



Figur 4.2: En illustration över hur daemonen LXD förhåller sig till övrig arkitektur.

LXD möjliggör bland annat nätverksstyrning av containrar via ett REST-api samt integrationsmöjligheter med orkestreringsverktyg som OpenStack [18][1]. Dessutom finns stöd för att hämta container-images från olika repositories.

LXD är skrivet i programspråket Go och stödjer officiellt nyare versioner av Linuxkärnan än 3.13. I övrigt är verktygets beroenden samma som för LXC.

Installation och konfiguration

LXD installerades först på Orange Pi. När LXD väl installerats och daemonen startats kan den konfigureras med kommandot lxd init, ett interaktivt konfigu-

[1]Se kapitel 5 för vidare diskussion kring orkestrering

reringsverktyg som låter användaren enkelt bestämma nätverksinställningar, filsystemspreferenser och dylikt.

Skapandet av en container är enkelt. En container-image, det vill säga rotfilsystem samt konfiguration, kan laddas ner från en server (antingen en som tillhandahålls av LXD eller en egen) samt startas med kommandot lxc launch. För att till exempel skapa en container med namnet guest-name baserad på BusyBox kan man använda lxc launch images:busybox guest-name. Det kan vara värt att betona att det finns säkerhetsrisker med att ladda ner rotfilsystem och konfigurationer från externa servrar. Det är förvisso smidigt men det kräver att det går att lita på servrarna och de personer som skapar de images man laddar ner. Säkerhetsriskerna går att mitigera genom användandet av en egen image-server så att full kontroll kan bibehållas.

När containern sedan är igång kan man köra kommandon i den eller starta en associerad tty. Utgående nätverk fungerar direkt men ingående behöver konfigureras likt för LXC *(se appendix 3)*.

På VG210 har LXD ett problem som gör verktyget helt obrukbart. Programspråket Go har egen minnes- och skräphantering och Go-program allokerar upp till 100 Mib virtuellt minne för dessa ändamål. VG210 har inte tillräckligt mycket ledigt minnesutrymme för detta och nästan alla Go-program kraschar direkt, så även LXD. Istället testades LXD på Beagle Bone-kortet där det fungerade felfritt, trots att Linuxkärnan på Beagle Bone är en äldre version än den som officiellt stöds av LXD.

Då LXD består av bland annat en daemon som alltid behöver köras innebär det ofrånkomligen att extra overhead introduceras. Under testning mättes daemonens fysiska minnesanvändning upp till ca 30 MiB detta utöver den overhead som introduceras av LXC. Konfigureringsmöjligheterna är samma som för LXC men det interaktiva konfigureringsverktyget och nätverksstyrningen förbättrar användarvänligheten.

4.2.3 Firejail

Firejail är ett program som likt många andra nyttjar Linuxfunktioner för att direkt köra applikationer i en container [19]. Firejail är med med andra ord ett applikationscontainersystem.

Utmärkande för Firejail är att verktyget inte kräver att man skapar något nytt

rotfilsystem över huvud taget. När en applikation startas via Firejail använder applikationen värdens filsystem. Vad i värdsystemet som syns och kan interageras med ifrån containern går enkelt att konfigurera via applikationsprofiler. Firejail är väldigt lättviktigt samt kräver en version av Linuxkärnan nyare än 3.0.

Installation och konfiguration

Installation av Firejail på Orange Pi var enkelt. När det har byggts är det ett antal binärfiler och bibliotek som behöver installeras. Därefter kan den applikation som ska sandboxas enkelt startas med kommandot firejail <applikationsnamn>. Nätverksanslutning konfigureras automatiskt; om inget annat specificeras så delar värd och container värdens nätverksanslutningar genom arv av värdsystemets nätverksnamespace. Mängder med andra nätvärkskonfigurationsmöjligheter finns dock. Firejail saknar stöd för checkpoint/restore.

En enkel sandbox-profil skapades. Denna går att finna i appendix 4. Testning av nätverkskommunikation skedde via en Apache webbserver som startades genom kommandot firejail –profile /etc/our_profile.profile httpd.

Firejail installerades sedan på VG210. Verktyget kräver likt andra implementationer att /run/ monteras som tmpfs. Dessutom behöver systemanropslistor för seccomp genereras, firejail kommer med verktyg för detta ändamål: fseccomp och fsec-optimize. Med detta gjort fungerade Firejail felfritt på VG210 samt även på Beagle Bone Black med samma inställningar.

Firejail använder sekundärminne av samma storleksordning som LXC (ca 9 MiB) men har fördelen att inte behöver ett rotfilsystem. Detta faktum kan man dra nytta av i system med brist på lagringsminne. Konfigurationsmöjligheterna är stora och sker antingen via kommandoradsargument eller likt LXC i en konfigurationsfil.

4.2.4 runC

runC – runContainer – är ett containeriseringsverktyg som implementerar OCI specifikationen och som används för starta, stoppa och hantera containrar [20]. runC är ett verktyg på liknande abstraktionsnivå som LXC. runC har sitt ursprung i Dockerarkitekturen och togs fram för att ersätta LXC. runC har senare blivit helt fristående från Dockerprojektet. runC är likt LXC en operativsystemscontaineriseringslösning. runC nyttjar ett eget bibliotek – libcontainer – för att implementera samma funktionalitet som LXC gör med liblxc.

runC har få beroenden. Rekommenderad Linuxkärneversion är minst 3.10, men runC fungerar med begränsad funktionalitet på alla versioner nyare än 2.6.2.

Installation och konfiguration

Att korskompilera runC var väldigt enkelt eftersom Go automatiskt kan hantera de flesta beroenden och har inbyggt stöd för korskompilering. Programmet består av en enda binärfil vilket gjorde att installationen med enkelhet kunde utföras genom att placera binärfilen samt använda bibliotek på lämplig plats.

En runC-container behöver innehålla två saker: en OCI-konfigurationsfil och ett rotfilsystem. Dessa extraherades för ett första test ur en nedladdad Dockerimage för en Apache webbserver. runC saknar inbyggt stöd för automatisk konfiguration av en nätvärksbrygga mellan värdsystem och container men ett tredjepartsverktyg, netns, finns för ändamålet [21]. netns läggs in som en s.k. hook (ett kommando som körs vid ett fördefinerat tillfälle, exempelvis före eller efter start av containern) i containerkonfigurationen och sätter upp en virtuell nätverksbrygga när containern startas.

Likt för LXC måste trafik vidarebefordras från värd till container. Se appendix 2 för ett exempel på hur en iptables konfiguration kan åstadkomma detta.

Då runC är skrivit i Golang lider det av samma problem på VG210 som övriga sådana implementationer: det kraschar vid start. På grund av detta testades runC i stället på Beagle Bone Black.

runC installerades på Beagle Bone tillsammans med det tidigare beskrivna verktyget netns. Likt övriga implementationer krävs en montering av /run som tmpfs. En config.json-fil skapades och placerades tillsammans med rotfilsystemet som skapats med Buildroot. Konfigurationen återfinns i appendix 1. CGroups-filsystemet

måste även monteras. LXC gör detta automatiskt men inte runC, så detta måste ske antingen via fstab eller via mount-kommandot. För att göra det så enkelt som möjligt användes ett bash-script som monterar alla cgroup-controllers som är aktiverade [22]. Containern startades sedan och fungerade felfritt.

runC är något större än LXC och Firejail lagringsmässigt, ca 30 MiB, men erbjuder liknande funktionalitet. Tekniken har stora konfigureringsmöjligheter och inställningar anges även här i en konfigurationsfil. Denna fil skrivs enligt OCI-specifikationen.

4.2.5 containerd

containerd är likt runC ett fristående projekt med ursprung i Dockerarkitekturen. containerd är en daemon som tillåter applikationer att smidigt implementera runC genom ett Golang-API [23]. I praktiken innebär detta att man höjer abstraktionsnivån genom göra det möjligt att styra runC via CRUD-operationerna. containerd innehåller dessutom funktionalitet för bland annat nätverkskonfiguration och gör det möjligt för utomstående processer att kommunicera med containrar. Eftersom runC terminerar efter att ha utfört en instruktion skapar containerd en så kallad shim till varje container. En shim är en process som existerar av tillförlitlighetsskäl och som ser till att containerprocesserna fortfarande går att kontrollera även om själva containerd-daemonen skulle terminera plötsligt.

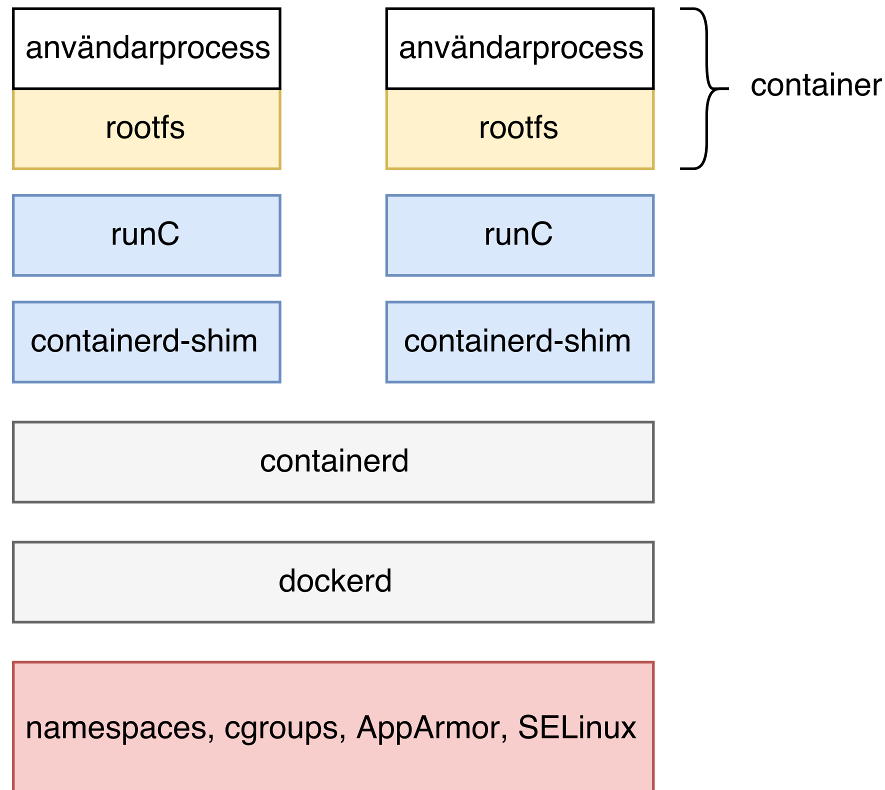
Utöver de beroenden containerd ärver från runC ställs också krav på en kompilator för Googles “protocol buffer” *(3.x)*.

Beslutet fattades att inte ta vidare containerd för testning. Detta då verktyget inte är designat för att direkt användas fristående utan genom integration med andra projekt via sitt Golang API. containerd bedöms vara ett mycket användbart verktyg om man är ute efter att designa en egen applikation som autonomt skall kunna skapa och hantera containrar. Utveckling av en sådan tjänst ansågs dock vara utanför omfånget av detta projekt.

4.2.6 Docker

Dockerprojektet är en containeriseringsplattform som likt LXD är på hög abstraktionsnivå. Docker är en applikationscontaineriseringslösning och dess huvudfokus är på deployment och användarupplevelse [24]. Docker åstadkommer detta genom

en daemon – dockerd – som kapslar in bland annat containerd och runC's funktionalitet.



Figur 4.3: En illustration över Dockerarkitekturen som visar sambandet mellan Linuxfunktionerna, dockerdaemonen, containerd samt runC.

Docker tillhandahåller hög containerportabilitet genom docker-images. Genom Dockers UX-funktionalitet kan man distribuera containeravbildningar via olika repositories. Docker möjliggör även för integration av en myriad relaterade verktyg och insticksmoduler som förenklar hanteringen av containrar. Exempel på sådana verktyg är Docker-compose: en modul för skapandet av multi-container applikationer samt orkestreringsverktyg som Docker Swarm [24] eller Kubernetes [25] för enklare deployment.

Installation och konfiguration

Om man bygger Docker statiskt (vilket rekommenderas av Docker-utvecklarna) består programmet endast av ett antal binärfiler vilket gör installationen väldigt

enkel. När det är installerat behöver man starta dockerd. Sedan kan man enkelt ladda ner och köra applikationscontainrar. Nätverk för containrar konfigureras automatiskt.

Docker testades först på Orange Pi. Kommandot: *docker run -dit -p 8080:80 "$PWD":flashcontainer-htdocs httpdarm32v7* var det enda som krävdes för att göra ett nätverkstest med en Apache webbserver. Kommandot fungerar under förutsättning att värdsystemet har internetaccess. En image innehållande Apache för rätt arkitektur laddas automatiskt ner från Dockers server. p-flaggan sätter upp en brygga från port 8080 på värden till port 80 på containern. PWD-argumentet sätter en miljövariabel i systemet på containern som används av Apache för att avgöra var dokumentroten för servern ska vara. Likt för LXD är det värt att nämna säkerhetsproblemen med att ladda ner images från externa servrar. Det är viktigt att man kan lita på innehållet i filerna man laddar ner och det är förmodligen önskvärt att skapa images manuellt eller att tillhandahålla en egen server.

Har man inte tillgång till internet kan man spara eller skapa images på en internetansluten dator och ladda in den manuellt på systemet utan internet via *docker save/load*.

Då Docker också är skrivit i Go och dessutom använder sig av runC uppstår samma problem med för lite fysiskt minne på VG210. Därför utfördes testning av Docker på Beagle Bone vilket fungerade felfritt. Docker består av ett par relativt resurstunga daemoner. Under testning mättes deras minnesanvändning i snitt upp till ca 60 MiB. Jämfört med LXD är detta ca dubbla mängden RAM. Dessa mätvärden är utöver den overhead som introduceras av Dockers användning av runC. Dockers konfigurationsmöjligheter är stora och Docker genererar dynamiskt en OCI-konfiguration vid körning. Genom Dockers 'Dockerfile'-system kan man dynamiskt skapa rotfilsystem-images. Har systemet tillgång till internet kan aktuella delar laddas ner samt installeras automatiskt.

4.2.7 rkt

rkt är likt runC ett containeriseringsverktyg som används för att köra OCI-containrar. Till skillnad från runC är flera av lågnivådetaljerna bortabstraherade med syftet att göra verktyget mer lättanvändligt [26]. rkt består av ett kommandoradsgränssnitt genom vilken man startar, stoppar och interagerar med containrar; det finns ingen daemon eller liknande. rkt utvecklades i samarbete med coreOS – distributionen 'container Linux' – för att bemöta kritiken Docker har fått om att vara svårintegrerat med init-system. Rkt har inte en hierarkisk arkitektur med flera

fristående deamon-processer likt Docker vilket löser vissa av dessa problem.

Även rkt är skriven i Go. För att köras kräver rkt även en Linuxkärna nyare än 3.18.

Installation och konfiguration

Ett antal försök att korskompilera och köra rkt genomfördes men misslyckades. Det saknas dokumentation kring huruvida rkt fungerar på ARM. Enligt den officiella dokumentation kräver dessutom rkt Linux 3.18+ samt systemd och bash för alla konfigurationer som inte bara är en enkel chroot-lösning [27]. Dessa beroenden är ej uppfyllda av distributionen som idag finns på VG210. En uppgradering av distributionen på VG210 hade naturligtvis varit möjlig men ansågs ligga utanför detta projekts omfattning. Om rkt önskas användas i framtiden rekommenderar vi att det undersöks till vilken grad tekniken tillhandahåller kompatibilitet för ARM samt om det finns några betydande fördelar som skulle motivera användandet av rkt.

4.3 Sammanfattning

Alla kandidater testades först på Orange Pi följt av på antingen VG210 eller Beagle Bone beroende på om kandidaten fungerade på VG210 eller ej. containerd samt rkt valdes bort från vidare testning med anledning av de iakttagelser som gjorts och problem som uppstått.

Tabell 4.1 ger en översikt över de olika implementationerna och vilka funktioner de har samt huruvida de har testats och fungerar på VG210 & Beagle Bone Black.

Tabell 4.1: Översikt över containeriseringsverktyg

	LXC	LXD	Firejail	runC	Docker	rkt
Nätverksisolation						
Filsystem- och mountisolation						
Begränsning av CPU & primärminnesanvändning						
Begränsning av sekundärminnesanvändning	[1]	[1]	[1]	[1]	[1]	[1]
Begränsning av nätverksbandbredd						
Begränsning av tillgång till hårdvaruenheter						
Kontroll över kapabiliteter						
Inbyggt stöd för checkpoint/restore						
Opriviligerade containrar						
seccomp						
Integration med SELinux						
Integration med AppArmor						
Separat rotfilsystem						
Automatiskt skapande av rotfilsystem						
Testad & fungerar på VG210						
Testad & fungerar på Beagle Bone						

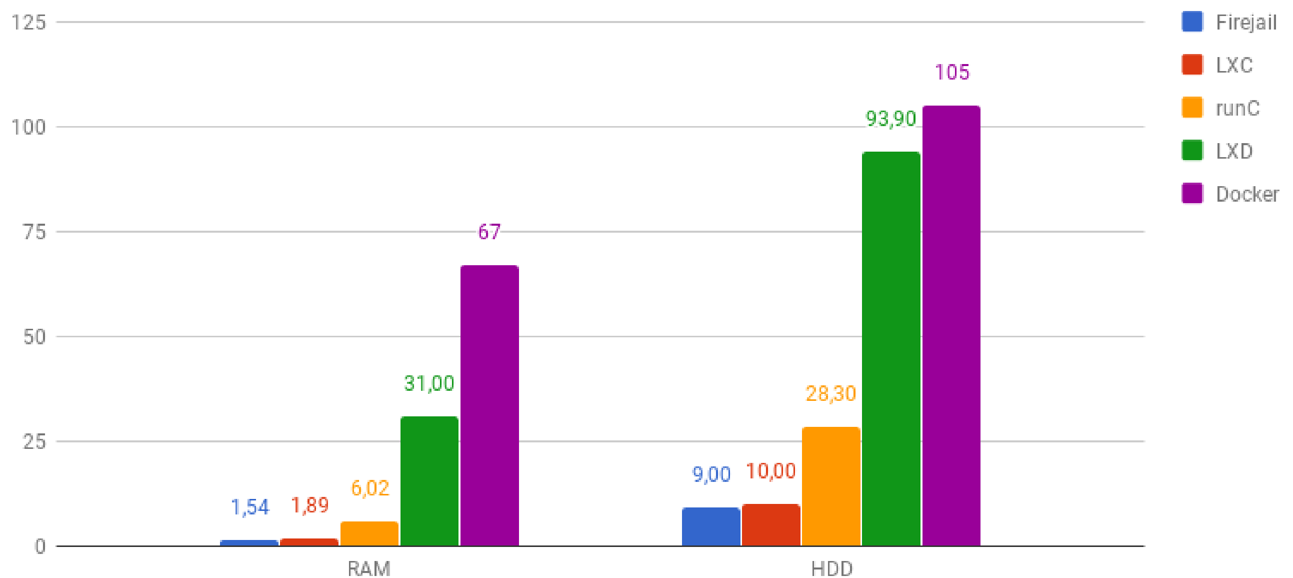
[1] Se diskussionskapitel

4.4 Benchmarking

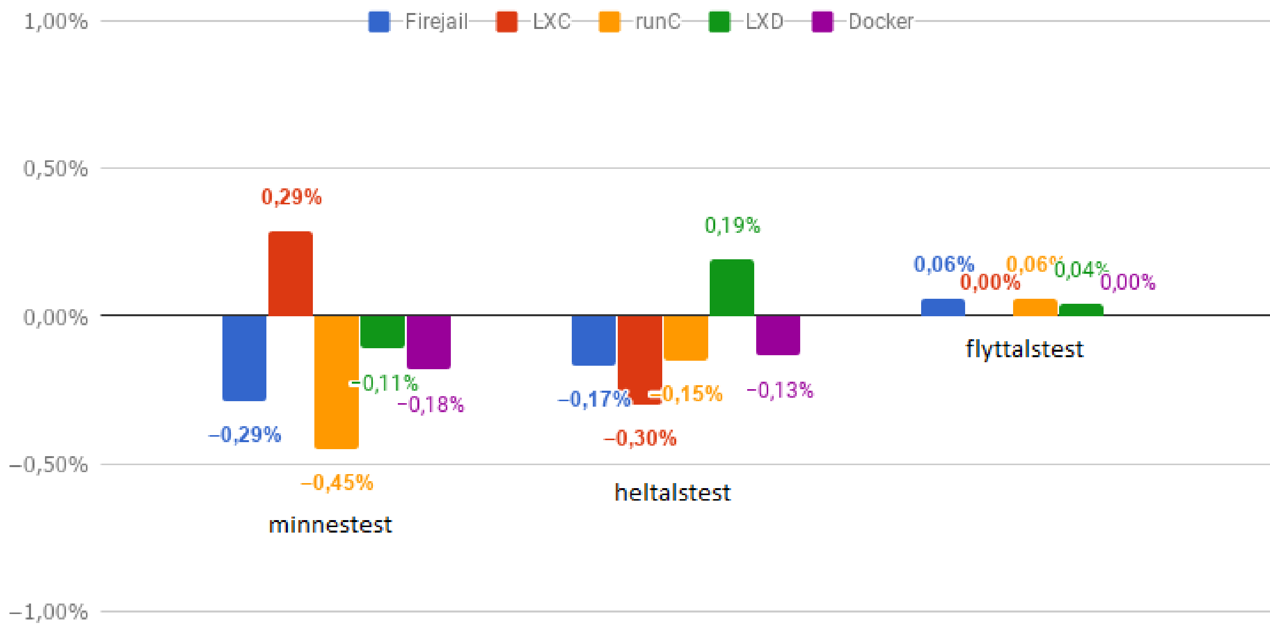
Nedan presenteras resultaten av den benchmarking som utfördes. Då runC behövde benchmarkas på Beagle Bone istället för VG210 genomfördes benchmarking på båda system. Resultaten mellan systemen skiljde sig inte åt på något betydande sätt så endast resultaten från Beagle Bone presenteras.

Figur 4.4 presenterar primärminnesanvändningen hos de olika implementationerna med en container startad körandes ett bash-skal. Resultaten för Docker och LXD inkluderar daemoner samt underliggande verktyg. Även sekundärminnesanvändning för installationen av de olika implementationerna presenteras. Rotfilsystem är ej inräknade. Docker använder mest resurser tätt följd av LXD, vilket är väntat då båda verktygen introducerar daemonprocesser. För övriga verktyg är resursanvändningen liknande men marginellt högre för runC. Värt att notera är att benchmarkingen är utförd med enbart en container aktiv. Hur minnesanvändningen ökar med antalet containrar beror på implementationen. Minnesanvändningen för daemonerna är konstant oavsett hur många containrar som körs. Hur minnesanvändningen ser ut med olika antal containrar igång har dock ej mätts.

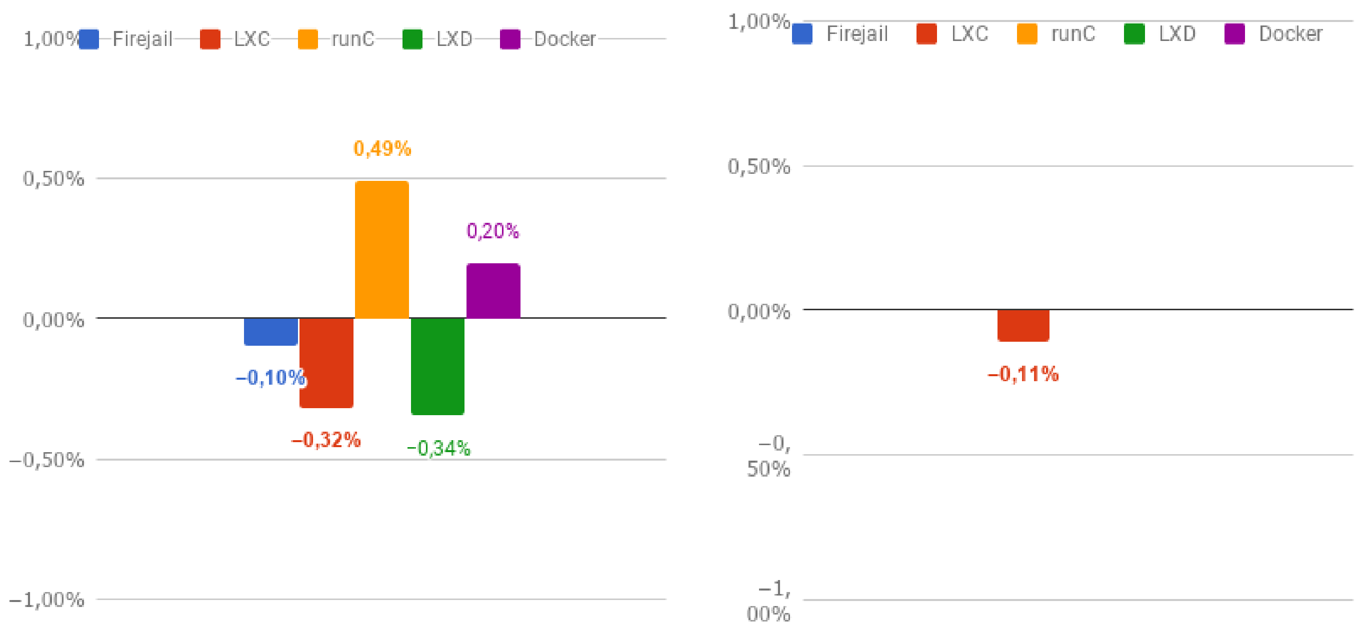
Figur 4.4: Primär- och sekundärminnesanvändning



Figur 4.5 visar den procentuella skillnaden mellan container och värdsystem i de fall benchmarking skett med nbench. Positivt värde innebär bättre resultat än på värdsystemet. nbench grupperar sina olika tester baserat på om de huvudsakligen testar minnesåtkomst, heltalsberäkning eller flyttalsberäkning och presenterar ett geometriskt medelvärde per kategori. Det är detta medelvärde som presenteras i figuren.



Figur 4.5: Procentuell skillnad i nbenchresultat med nbench körandes inuti container jämfört med utan container. Geometriska medelvärden per olika testtyper.



Figur 4.6: Resultat diskbenchmarking samt nätverksbenchmarking

Figur 4.6 visar till vänster resultatet av benchmarkingen av skrivhastighet till disk som gjorts med verktyget dd. Till höger visas resultaten från benchmarking av nätverkskommunikation gjort med verktyget iperf. Även i dessa två fall innebär positivt värde bättre resultat än värdsystemet.

Resultaten indikerar att ingen implementation introducerar någon signifikant processor- nätverk- eller diskoverhead samt att prestandan är nära identisk implementationerna sinsemellan. Skillnader på plus/minus en halv procent likt de som visats i grafen bedöms vara inom felmarginalen för testverktygen. Till exempel kan man med logisk säkerhet utgå ifrån att en container inte har bättre prestanda än värdsystemet vilket leder till att vi anser att resultaten ligger inom statistisk felmarginal.

4.5 Funktionalitetstester

I slutskedet av projektet testades användning av olika gränssnitt som skulle kunna tänkas vara lämpliga att synliggöra inom en container på VG210. Detta kapitel redogör för resultatet av dessa tester.

SSH

En SSH-server testades inuti en LXC container tillsammans med vidarebefodring av SSH-trafik från värd. Denna konfiguration gör att alla SSH-anslutningar till värd blir anslutningar till containern. Anslutning är därmed sandboxad och det går inte att via SSH nå delar av systemet som inte existerar i containern. Vi utgår ifrån att det fungerar på samma sätt på andra implementationer än LXC.

CAN

Det finns en standarddrivrutin för CAN-bussar i Linuxkärnan, SocketCAN. I SocketCAN är CAN implementerat i den vanliga nätverksstacken och används mer eller mindre som ett vanligt Ethernetgränssnitt [6]. Därmed innefattas SocketCAN-gränssnitt i ett nätverksnamespace. Den vanliga lösningen för att skicka trafik mellan nätverksnamespaces, bryggor, fungerar dock inte för SocketCAN. Det finns stöd för virtuella CAN-gränssnitt men ej för bryggning. Det fysiska gränssnittet kan enbart tillhöra ett namespace åt gången vilket i praktiken innebär att container och värd måste tillhöra samma namespace om SocketCAN inom containern ska vara möjligt. Detta medför säkerhetsrisker i form av att hela nätverksstacken på värdsystemet synliggörs i containern.

På VG210 finns även två CAN-gränssnitt anslutna till en separat I/O processor. Dessa gränssnitt kommunicerar för nuvarande med program på huvudprocessorn via IPC-socketfiler som är placerade under /tmp. För att ge containrar tillgång till dessa gränssnitt räcker det med att göra /tmp tillgänglig i containern *(se exempelkonfiguration för LXC i appendix 3)*. Görs filerna ej tillgängliga är inte heller CAN-gränssnitten tillgängliga. Tyvärr verkar det inte finnas någon metod för att på detta sätt ge endast läsbehörighet och inte skrivbehörighet till CAN-bussen vilket gör att detta inte är en optimal lösning. Möjliga lösningar på CAN-problemet presenteras i diskussionskapitlet.

5 Diskussion

I detta kapitel diskuteras varför vissa kandidater bedöms vara mer lämpliga än andra och vad som faktiskt skiljer de olika kandidaterna åt i praktiken. Även funktionalitet- och säkerhetsaspekter diskuteras.

5.1 Dagens generation av VG210

Under testfasen på VG210 upptäcktes det tidigt att de stora minnesbegränsningarna på systemet tillsammans med hur Golang är implementerat gjorde det omöjligt att få en delmängd av de olika kandidaterna att fungera. Detta ledde till en viss splittring mellan de kandidater som är möjliga på dagens generation av VG210 samt de som eventuellt är lämpliga på en framtida version. Detta avsnitt redogör för de lösningar som är möjliga i nuläget och nästföljande avsnitt redogör för möjligheter som finns på en förbättrad gateway.

De kandidater som är möjliga på nuvarande gateway är LXC samt Firejail. Teknikerna är lika varandra på många sätt. De är båda skrivna i C, har försumbar overhead samt mycket lika konfigurationsmöjligheter. Även om det finns en uttalad skillnad mellan dem i att Firejail är designat som en applikationscontainerlösning och LXC som en operativsystemscontainerlösning är båda verktyg flexibla. Det går att använda LXC för att köra en enstaka applikation och Firejail för att skapa en operativsystemscontainer trots att det blir mer komplicerat.

Den avgörande skillnaden mellan LXC och Firejail är huruvida man vill använda sig av en lösning som bygger på att containern har ett separat rotfilsystem eller ej. Eftersom Firejail delar värdens rotfilsystem kräver detta att man installerar applikationen man vill containerisera samt dess beroenden direkt på värdsystemet. En fördel med en sådan lösning är att hårddiskanvändningen blir mindre.

Användning av ett separat rotfilsystem som i fallet LXC ökar containerns portabilitet enormt i de fall man har en mer komplex applikation som till exempel har beroenden i form av en Java-exekveringsmiljö eller bibliotek som måste installeras tillsammans med applikationen. Ett separat rotfilsystem kan även ha logistiska fördelar om man har många olika containrar på samma värdsystem och man önskar tydligare översikt över vilka beroenden som hör till vilken applikation. Med operativsystemscontainersering är det även möjligt att ge en tredjepartsanvändare tillgång enbart till en container. Detta kan implementeras med en SSH-brygga

som beskrivits tidigare. Om processer i containern bara har tillgång till de resurser de behöver och kan använda på ett säkert sätt går det att låta tredjepartsanvändare installera och köra vilken mjukvara de önskar utan säkerhetsimplikationer. Denna lösning blir mer som ett full virtuellt system men utan overhead.

5.2 Framtidens generation av VG210

Vid en utökning av minnesutrymmet på en framtida gateway kan de alternativ som inte var möjliga på dagens generation tas i beaktande. Ett avgörande ställningstagande är vad man i framtiden önskar kunna göra med containrar som inte uppfylls av lösningarna som fungerar i dagsläget samt om tekniken man väljer kan skalas efter dessa framtida behov.

Ett tänkbart framtida användningsområde är möjligheten att automatisera underhållet av en större mängd VG210 enheter som till exempel en fordonsflotta. Sådan automation skulle höja abstraktionskraven på containeriseringsimplementationen som används samt ställa nya krav på extern styrning och möjligheter till integration med andra verktyg. I nästkommande stycke kommer några alternativ för sådan automatisering att nämnas.

Orkestrering

Orkestrering är ett begrepp som behandlar möjligheterna att från en central punkt, en huvudnod, distribuera uppdateringar eller andra förändringar till en skalbar mängd enheter [28]. Behovet av orkestrering är sprunget ur problemen framgångsrika webbtjänster ställs inför när användartalet plötsigt ökar.

Utöver Dockers egna orkesteringsmöjligheter genom Docker-swarm som installeras tillsammans med Docker-motorn erbjuder Docker integreringsmöjligheter orkestreringsverktyget Kubernetes: ett mycket populärt open-source projekt ursprungligen utvecklat av Google som har utbredd användning inom webbindustrin [25]. LXD har inga inbyggda orkestreringsmöjligheter men erbjuder integrering med OpenStack, en samling verktyg framtagna för att underlätta utveckligen av molnstjänster som bland annat inkluderar orkestringingsverktyget heat [18].

Fler alternativ till orkestrering finns att utforska. Däribland finns libvirt, ett administrationsverktyg genom vilket det via nätverkskommunikation går att hantera

virtuella maskiner och containrar. Libvirt går att använda direkt med LXC via en speciell drivrutin [29].

5.3 Säkerhetsaspekter

Alla containeriseringskandidater som behandlats i detta projekt har mycket likartad säkerhetsfunktionalitet då de är baserade på samma säkerhetsfunktioner i Linuxkärnan. Resurser så som CPU, nätverksbandbredd och minne går att begränsa med funktioner som CGroups och rlimit. På så visar hindras användare av sandboxen från att störa ut den primära funktionaliteten på VG210. Tillgång till fysiska gränssnitt som Ethernet, CAN eller serieportar kan begränsas på liknande vis genom cgroups och mount-namespace. Känslig data kan skyddas genom att man styr vilka delar av filsystemet användare av sandboxen kan se. Detta görs via namespaces och chroot. Vilka systemanrop som får göras och vilka kapabiliteter en användare i containern har går att styra. Som tidigare nämnt är det i regel en god idé att släppa på CAP_SYS_ADMIN och andra kapabiliteter i containern så att det inte går att montera nya delar av filsystemet inifrån containern och dylikt. En god idé är att släppa alla kapabiliteter och sedan vitlista vid behov.

Värt att notera är att den säkerhet som containeriseringstekniken tillhandahåller bygger till stor del på användadet av namespaces för att begränsa synligheten hos delar av operativsystemet. En konsekvens av detta är att högre säkerhet resulterar i mer komplicerad kommunikation vilket är per design.

Möjligheter till utökad säkerhet

Ser man skäl till att ha hårdare säkerhetskrav än vi har haft under våra test finns det ytterligare möjligheter. Främst handlar ökad säkerhet om att begränsa containrars behörighet så att inte värdsystemet skulle äventyras om en användare mot förmodan bryter sig ut ur containern. Alla kandidater – utom Firejail vars implementation skiljer sig åt – har stöd för opriviligerade containrar. Som tidigare nämnt skyddar denna teknik värdsystemet om en process mot förmodan bryter sig ut från en container. Opriviligerade containrar kräver usernamespaces, samt vissa användarrymdsverktyg som tekniken är beroende av.

Vi bedömer inte att det är speciellt sannolikt att man på VG210 skulle vilja gå till medvetet angrepp på en containerprocess för att få rottillgång till värden, vilket är

det som opriviligerade containrar kanske främst skyddar mot. Detta då systemet i regel bara är direkt tillgängligt för en begränsad mängd användare. Vill någon däremot köra tjänster i en container som är direkt tillgängliga för slutkunder (passagerare eller dylikt) är det rimligt att enbart använda opriviligerade containrar.

AppArmor/SELinux

AppArmor/SELinux tillhandahåller som beskrivet ytterligare ett behörighetslager och det är möjligt att använda dessa verktyg tillsammans med containeriseringsverktygen för att öka säkerheten om så önskas. AppArmor & SELinux kan vara viktiga om stöd för opriviligerade containrar saknas men är mer eller mindre irrelevant om usernamespaces används korrekt. Värt att nämna är att de nyare versionerna av containeriseringsverktygen i regel har inbyggt stöd för de säkerhetsregler som är möjliga med SELinux och AppArmor [9]. SELinux & AppArmor aktiveras på hela värdsystemet, inte bara inom containern, och kräver dessutom extra användarrymdsverktyg.

CAN-trafik

Som funktionstesterna visade fanns det vissa problem med CAN och vidarebefodring till containrar. Det finns verktyg för att skicka CAN-ramar via TCP [30] eller MQTT vilket skulle kunna användas för att vidarebefordra CAN-trafik till och eventuellt från containrar. En sådan lösning kräver mer mjukvara men erbjuder en väldigt hög grad säkerhet och kontroll över CAN-trafiken till och från containrar. Samma typ av lösning kan vara lämplig för andra kommunikationsgränssnitt och protokoll.

Diskkvoter

En funktionalitet som de testade containeriseringsverktygen inte direkt erbjuder[1] är möjligheten att skapa diskkvoter, det vill säga att begränsa hur mycket sekundärminne en container får använda. Det kan vara lämpligt att ha diskkvoter för att se till att containrar inte kan störa värdsystemet genom att använda allt lagringsminne. Detta kan implementeras på olika sätt; till exempel kanske containrar

[1] Docker har delvis stöd för diskkvoter om man använder specifika filsystem.

bör placeras på en egen partition. Alternativt finns det filsystemsimplementationer som har stöd för diskkvoter, till exempel btrfs [31].

5.3.1 Etik & Miljöaspekter

Eftersom Pilotfish är ett företag vars arbete inriktar sig på kollektivtrafiksindustrin är det av etiska skäl viktigt med säkra system, då brister kan komma att påverka intet ont anande passagerare och medföra samhälleliga kostnader. Detta projekt för med sig en etisk vinst i hänseendet att vi visat att sandboxing är högst tillämpligt på system så som detta och genom att projektet bidrar till en ökad kunskap om vilka säkerhetsimplikationer som finns. Detta projekt bedömer vi även kan komma att bidra till en miljömässig vinst. Vid överförande av mjukvara från extern hårvara till Pilotfish Vehicle Gateway minskar det totala kravet på fysiska resurser i de fordon som berörs.

6 | Slutsats & Rekommendation

Under projektets gång har sju olika kandidater utvärderats och deras egenskaper fastställts. I detta kapitel redovisas de slutsatser som dragits efter testning av kandidaterna på våra testsystem. Hänsyn har tagits till flera utvärderingskriterier. Benchmarking och andra mätningar har utförts som resulterat i mätvärden på primärminnesanvändning, processoroverhead samt sekundärminnesanvändning. Även mjukare kriterier så som användarvänlighet och konfigurationsmöjligheter har diskuterats. Alla kandidater erbjuder i slutändan liknande och fullgoda möjligheter för sandboxing. Således är målet att förhindra negativ inverkan från tredjepartsmjukvara mött oavsett vilken implementation som i slutändan används. Dock är de implementationer som är skrivna i Golang problematiska att använda på dagens version av VG210 och en större mängd resurser. Alla implementationer erbjuder övervakning av processer (lokalt eller via nätverk) samt migration av containrar.

Även ur ett säkerhetsperspektiv är implementationerna funktionellt väldigt lika; det ingen implementation som har någon relevant säkerhetsfunktionalitet som andra saknar. Alla implementationer bedöms ha tillräckliga säkerhetsfunktioner för de användningsområden som är tilltänkta. Därmed är valet av implementationen oberoende av även detta kriterium.

Rekommendation

Likheterna mellan implementationerna till trots rekommenderar vi att LXC används till dagens gateway. Med en lösning mer lik en traditionell virtuell maskin kan en kund direkt få tillgång till en säker miljö där de kan installera vilken mjukvara de än önskar vilket vi tror kan vara ett välfungerande upplägg. I egenskap av att vara en implementation med fokus på operativsystemcontainerisering anser vi att LXC är tekniken för att åstadkomma detta. Dessutom fungerar LXC väldigt bra även på dagens version av VG210, har goda konfigurationsmöjligheter samt använder väldigt lite resurser. Det går även att relativt smidigt använda LXC som applikationscontaineriseringslösning om man så önskar.

För framtidens gateway rekommenderar vi även här LXC. Önskas enklare konfiguration och mer funktionalitet är det enkelt att i efterhand integrera LXC-containrar med LXD. LXC/LXD bedömmer vi också ha de förutsättningar som krävs för en eventuell integration med något av de orkestreringsverktyg som nämnts.

Skulle man vilja använda Docker - till exempel för att man har mer erfarenhet av det - så är även detta fullt möjligt på en gateway med mer primärminne. Framtidens gateway planeras kunna ha 512 MiB RAM vilket vi bedömer ge goda förutsättningar för sandboxing med även lösninger som Docker samt för en eventuell utbyggnad med deployment och orkestrering.

Eftersom ingen containeriseringsimplementation har någon betydande CPU-overhead har vi inga särskilda rekommendationer gällande val av processor.

Litteraturförteckning

[1] “BusyBox”. [Online]. Tillgänglig på: https://busybox.net/about.html. [Hämtad: 16-Maj-2018].

[2] T. Olzak, A. Bawcom, och J. Hoopes, Virtualization for Security. Boston, MA: Syngress, 2008.

[3] J. E. Smith och R. Nair, Virtual machines: versatile platforms for systems and processes. San Francisco, CA: Morgan Kaufmann, 2005.

[4] S. Kumaran. S., Practical LXC and LXD - Linux Containers for Virtualization and Orchestration. Berkeley, CA: Apress, 2017.

[5] Open Container Initiative, “Open Container Initiative (OCI) Releases v1.0 of Container Standards”, 2017. [Online]. Tillgänglig på: https://www.opencontainers.org/announcement/2017/07/19/open-container-initiative-oci-releases-v1-0-of-container-standards [Hämtad: 10-Maj-2018].

[6] The Linux Kernel Foundation, “The Linux Kernel documentation”, 2018. [Online]. Tillgänglig på: https://www.kernel.org/doc/html/latest/. [Hämtad: 19-April-2018].

[7] Borate, I. and K., R. (2016). Sandboxing in Linux: From Smartphone to Cloud. International Journal of Computer Applications, 148(8), pp.1-8.

[8] Schreuders, Z., McGill, T. and Payne, C. (2011). Empowering End Users to Confine Their Own Applications. ACM Transactions on Information and System Security, 14(2), pp.1-28.

[9] J. Hertz, Abusing privileged and unprivileged linux containers. (Whitepaper). NCC Group, 2016.

[10] Virtuozzo Inc, “CRIU”. [Online]. Tillgänglig på: https://criu.org. [Hämtad: 17-Maj-2018].

[11] “Orange Pi”. [Online]. Tillgänglig på: http://www.orangepi.org/. [Hämtad: 16-Maj-2018].

[12] Beagle Board, “BeagleBone Black”. [Online]. Tillgänglig på: https://beagleboard.org/black. [Hämtad: 16-Maj-2018].

[13] Uwe F. Mayer, “nbench”. [Online]. Tillgänglig på: https://www.math.utah.edu/ mayer/linux/bmark.html [Hämtad: 19-April-2018].

[14] IEEE & The Open Group, “dd”, 2018. [Online]. Tillgänglig på: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/dd.html [Hämtad 16-Maj-2018].

[15] ESNet Software, “iperf3 3.5 documentation”, 2018. [Online]. Available: https://software.es.net/iperf/. [Hämtad: 19-April-2018].

[16] Linaro, “About Linaro”, 2018. [Online]. Available: https://www.linaro.org/about/. [Hämtad: 24-April-2018].

[17] “Buildroot - Making Embedded Linux Easy”. [Online]. Tillgänglig på: https://buildroot.org/. [Hämtad: 16-Maj-2018].

[18] OpenStack.org, “Open source software for creating private and public clouds”. [Online]. Tillgänglig på: https://www.openstack.org/. [Hämtad: 16-Maj-2018].

[19] “Firejail”. [Online]. Tillgänglig på: https://firejail.wordpress.com/. [Hämtad: 16-Maj-2018].

[20] OpenContainers, “GitHub: opencontainers/runc”. [Online]. Tillgänglig på: https://github.com/opencontainers/runc. [Hämtad: 16-Maj-2018].

[21] Genuinetools, “GitHub: genuinetools/netns”. [Online]. Tillgänlig på: https://github.com/genuinetools/netns. [Hämtad: 16-Maj-2018].

[22] Tianon, “GitHub: tianon/cgroupfs-mount”. [Online]. Tillgänlig på: https://github.com/tianon/cgroupfs-mount. [Hämtad: 16-Maj-2018].

[23] Containerd.io, “containerd”. [Online]. Tillgänglig på: https://containerd.io/ [Hämtad 14-Maj-2018].

[24] Docker Inc, “Docker”. [Online]. Tillgänglig på: https://www.docker.com/. [Hämtad: 16-Maj-2018].

[25] Kubernetes.io, “Production-Grade Container Orchestration”. [Online]. Till-

gänglig på: https://kubernetes.io/. [Hämtad: 16-Maj-2018].

[26] CoreOS, “CoreOS Blog”. [Online]. Tillgänglig på: https://coreos.com/rkt/. [Hämtad: 16-Maj-2018].

[27] CoreOS, “Build Configuration”. [Online]. Tillgänglig på: https://coreos.com/rkt/docs/latest/build-configure.html [Hämtad 14-Maj-2018].

[28] T. Hunter, Advanced microservices: a hands-on approach to microservice infrastructure and tooling. Berkeley, CA: Apress, 2017.

[29] “libvirt: LXC container driver”. [Online]. Tillgänglig på: https://libvirt.org/drvlxc.html. [Hämtad: 16-Maj-2018].

[30] dschanoeh, “GitHub: dschanoeh/socketcand”. [Online]. Tillgänglig på: https://github.com/dschanoeh/socketcand. [Hämtad: 14-Maj-2018].

[31] “btrfs Wiki”. [Online]. Tillgänglig på: https://btrfs.wiki.kernel.org. [Hämtad: 17-Maj-2018].

A Appendix

A.1 OCI-konfiguration

De delar som ändrades från den av runC automatgenererade OCI-konfigurationen är följande.

```
[...]
  "hooks": {
    "prestart": [
      {
        "path":"/opt/runc/netns-linux-arm",
        "args":["netns-linux-arm"]
      },
      {
        "path":"/bin/sh",
        "args":["sh", "/opt/runc/cgroup_mount.sh"]
      }
    ]
  }
```

A.2 Iptables konfiguration

Följande är ett exempel på en konfiguration av iptables för vidarebefodring av paket från värd till container och vice versa. I exemplet har både container och värd virtuella gränssnitt som är sammankopplade med en nätverksbrygga. Ip-address för det virtuella gränssnittet i containern är 10.10.20.10 och på värden 10.10.20.1. Det externa gränssnittet på värden har namnet eth0. Trafik på port 22 vidarebefodras. Följande kommandon exekveras på värd.

```
# aktivera ip forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward

# för att containern ska kunna kommunicera med internet
# krävs att paket som kommer från containern ser ut att
# komma i från värdens eth0-ipaddress (dvs. har samma source ip).
# Detta åstadkoms med masquerading.
iptables -t nat -A POSTROUTING -s \
10.10.20.1/255.255.255.0 -o eth0 -j MASQUERADE

# tillåt forwarding från det externa gränsnittet eth0 till
# bryggan br0 och v.v.
iptables -A FORWARD -i br0 -o netns0 -j ACCEPT
iptables -A FORWARD -o br0 -i netns0 -j ACCEPT

# skicka vidare alla paket på tcp port 22 till containern
iptables -t nat -A PREROUTING -p tcp --dport 22 -j DNAT \
--to-destination 10.10.20.10:22

# tillåt att alla redan etablerade anslutningar på port 22 skickar
# vidare paket till och från containern
iptables -A FORWARD -p tcp -d 10.10.20.10 --dport 22 \
-m state --state NEW,ESTABLISHED,RELATED -j ACCEPT
```

A.3 LXC-konfiguration

```
## Nätverksinställingar för enkel brygga
lxc.net.0.type = veth
lxc.net.0.flags = up
lxc.net.0.name = veth0
lxc.net.0.ipv4.address = 10.10.20.10/24
lxc.net.0.ipv4.gateway = 10.10.20.1
lxc.net.0.link = br0

# själv bryggan måste skapas manuellt på värden
# med exempelvis brctl
# brctl addbr br0
# tilldela bryggan samma ip som konfigurerats ovan för
# containerns gateway
# ifconfig br0 10.10.20.1 netmask 255.255.255.0

# Generella kontainerinställningar

lxc.rootfs.path = dir:/<PATH_HERE>/rootfs
# Hostname
lxc.uts.name = vg210ng

# Släng alla capabilities, vitlista vid behov
lxc.cap.keep = none

lxc.signal.halt = SIGUSR1
lxc.signal.reboot = SIGTERM
lxc.tty.max = 4
lxc.tty.dir = lxc
lxc.pty.max = 1024

# Vitlista cgroups genom att sätta "default deny all"
lxc.cgroup.devices.deny = a
## Tillåt skapandet av device-noder
lxc.cgroup.devices.allow = c *:* m
lxc.cgroup.devices.allow = b *:* m
## Tillåt specifika devices
### /dev/null
lxc.cgroup.devices.allow = c 1:3 rwm
### /dev/zero
lxc.cgroup.devices.allow = c 1:5 rwm
```

```
### /dev/full
lxc.cgroup.devices.allow = c 1:7 rwm
### /dev/tty
lxc.cgroup.devices.allow = c 5:0 rwm
### /dev/console
lxc.cgroup.devices.allow = c 5:1 rwm
### /dev/ptmx
lxc.cgroup.devices.allow = c 5:2 rwm
### /dev/random
lxc.cgroup.devices.allow = c 1:8 rwm
### /dev/urandom
lxc.cgroup.devices.allow = c 1:9 rwm
### /dev/pts/*
lxc.cgroup.devices.allow = c 136:* rwm
### fuse
lxc.cgroup.devices.allow = c 10:229 rwm

# Monteringspunkter
lxc.mount.auto = cgroup:mixed proc:mixed sys:mixed
lxc.mount.entry = /sys/fs/fuse/connections
    sys/fs/fuse/connections none bind,optional 0 0

# För tillgång till LPC-CAN
# Om mappar på hosten är bind-monterade som endast läsbara
# i containern bör sys_admin alltid släppas, annars
# går det enkelt att montera om som skrivbar i containern
lxc.mount.entry = /tmp tmp none bind,nosuid,ro 0 0

lxc.mount.entry = /dev/vg200ng dev/vg200ng none
    create=dir,bind,nosuid,ro 0 0

# Svartlista ett antal syscalls med seccomp.
lxc.seccomp.profile = /usr/share/lxc/config/common.seccomp
```

A.4 Firejail-konfiguration

```
# brygga ihop värdsystemets interface br0 med containern
# sätt containerns ip till 10.10.20.10 och gateway 10.10.20.1
# bryggan måste skapas manuellt med exempelvis brctl

net br0
ip 10.10.20.10
defaultgw 10.10.20.1

# monterar /root och /home i containern som tmpfs med endast
# minimala nödvändiga resurser tillgängliga
private
# alla ändringar till filsystemet slängs därmed
# när containern avslutar

# slänger alla capabilities i containern, vitlista vid behov
caps.drop all

# ser till att det inte går att få nya privilegier
# i containern genom att köra ett suid-program
nonewprivs

# ge _endast_ tillgång till mappen /flash
whitelist /flash

# alternativt går det att svartlista specifika mappar
# blacklist /tmp

# blockera systemanrop, mkdir som trivialt exempel
seccomp.drop mkdir
```

A.5 Kompileringsflaggor för Linuxkärnan

Följande funktioner måste vara aktiverade vid kompilering av Linuxkärnan för att samtliga testade containeriseringsimplementationer ska fungera.

CONFIG_SECURITY_APPARMOR
CONFIG_NAMESPACES
CONFIG_USER_NS
CONFIG_NET_NS
CONFIG_PID_NS
CONFIG_IPC_NS
CONFIG_UTS_NS
CONFIG_CGROUPS
CONFIG_CGROUP_CPUACCT
CONFIG_CGROUP_DEVICE
CONFIG_CGROUP_FREEZER
CONFIG_CGROUP_SCHED
CONFIG_CPUSETS
CONFIG_MEMCG
CONFIG_KEYS
CONFIG_VETH
CONFIG_BRIDGE
CONFIG_BRIDGE_NETFILTER
CONFIG_NF_NAT_IPV4
CONFIG_IP_NF_FILTER
CONFIG_IP_NF_TARGET_MASQUERADE
CONFIG_NETFILTER_ADVANCED
CONFIG_NETFILTER_XT_MATCH_ADDRTYPE
CONFIG_NETFILTER_XT_MATCH_CONNTRACK
CONFIG_NETFILTER_XT_TARGET_CHECKSUM
CONFIG_NETFILTER_XT_MATCH_COMMENT
CONFIG_NETFILTER_XT_MATCH_IPVS
CONFIG_IP_NF_NAT
CONFIG_NF_NAT
CONFIG_NF_NAT_NEEDED
CONFIG_POSIX_MQUEUE
CONFIG_DEVPTS_MULTIPLE_INSTANCES
CONFIG_MACVLAN
CONFIG_VLAN_8021Q
CONFIG_OVERLAY_FS

Om checkpoint/restore-funktionalitet önskas krävs även dessa

CONFIG_CHECKPOINT_RESTORE
CONFIG_FHANDLE
CONFIG_EVENTFD
CONFIG_EPOLL
CONFIG_UNIX_DIAG
CONFIG_INET_DIAG
CONFIG_PACKET_DIAG
CONFIG_NETLINK_DIAG

A.6 Inittab konfiguration

```
::sysinit:/bin/hostname -F /etc/hostname
::sysinit:/etc/init.d/rcS

# krävs för att ge en terminal när containern startas
console::askfirst:/bin/sh

::shutdown:/etc/init.d/rcK
```