Dissertation Type: software development

DEPARTMENT OF COMPUTER SCIENCE

Building a Testbed for Evaluating Privacy Enhancing Technologies (PETs)

Jacob Daniel Halsey

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Friday 21st May, 2021

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of BSc in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Jacob Daniel Halsey, Friday 21st May, 2021

Contents

Executive Summary

The goal of this project is to produce a simple and lightweight testbed platform for evaluating privacy enhancing technologies. It should provide support for testing varied architectures and network topologies, such as client-server and peer-to-peer applications. It should also support simulating applications for different types of platforms including mobile phone apps.

Summary of work:

- I have developed a flexible command line tool called *kvm-compose* for Linux using the Rust language and *libvirt* library for building and destroying virtual testing environments.
- In the process I have made some contributions to open source libraries including `libvirt-rust` (the *Rust* language bindings to *libvirt*).
- I have then implemented some example projects using the testbed tool.

Supporting Technologies

- *Linux KVM* (Kernel-based Virtual Machine) - https://www.linux-kvm.org/
- *Open vSwitch* Virtual multilayer switch - https://www.openvswitch.org/
- *libvirt* Virtualization API - https://libvirt.org/
- *Rust* Language, Compiler, Toolchain, etc. - https://www.rust-lang.org/
- *libvirt-rust* Rust bindings to the libvirt - https://gitlab.com/libvirt/libvirt-rust
- *clap* Rust command Line Argument Parser - https://github.com/clap-rs/clap
- *serde* Rust Serialization framework - https://github.com/serde-rs/
- *serde-yaml* YAML backend for serde - https://github.com/dtolnay/serde-yaml
- *serde-plain* Plain text backend for serde - https://github.com/mitsuhiko/serde-plain
- *thiserror* Rust error derive macro - https://github.com/dtolnay/thiserror
- *anyhow* Rust error handling framework - https://github.com/dtolnay/anyhow
- *simple_logger* Rust logging implementation - https://github.com/borntyping/rust-simple_logger
- *xml-rs* XML library for Rust - https://github.com/netvl/xml-rs
- *validator* Rust struct validation - https://github.com/Keats/validator
- *directories* User data directories library - https://github.com/dirs-dev/directories-rs
- *reqwest* Rust HTTP Client - https://github.com/seanmonstar/reqwest
- *indicatif* Rust command line progress indicator - https://github.com/mitsuhiko/indicatif
- *tempfile* Rust temporary file library - https://github.com/Stebalien/tempfile
- *casual* Rust user input parser - https://github.com/rossmacarthur/casual
- *derive-new* Rust new constructor macro - https://github.com/nrc/derive-new
- *enum-iterator* Rust macro for iterating enums - https://github.com/stephaneyfx/enum-iterator
- *rust-embed* Embeds files into Rust binaries - https://github.com/pyros2097/rust-embed

Acknowledgements

I would like to thank my supervisor Professor Awais Rashid and co-supervisor Joe Gardiner for their project proposal and support and guidance in completing it.

Chapter 1

Contextual Background

1.1 What are Privacy Enhancing Technologies (PETs)?

Before we discuss Privacy Enhancing Technologies we must consider what we mean by 'privacy". In his 2006 article D. J. Solove *A Taxonomy of Privacy* [1] notes that the definition of privacy has often been very broad or vague, and therefore sets out to develop a taxonomy of privacy violations. He has defined four groups of harmful activities (See figure 1.1).

Broadly speaking a Privacy Enhancing Technology is any solution or approach in hardware or software that helps protect a user from such privacy violations [2]. Some examples of PETs could include Onion routing such as the Tor network (which enables anonymous communication), or end-to-end encrypted messaging systems such as the Signal protocol. Kaaniche et al. [3] have defined a more comprehensive classification of PETs (see Figure 1.2).

1.2 Importance of Evaluating PETs

1.3 REPHRAIN

The UK Research and Innovation (UKRI) is a non-departmental public body of the United Kingdom Government sponsored by the Department for Business, Energy and Industrial Strategy [4]. In October 2020 the UKRI announced the creation of the National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online (REPHRAIN) [5]. The centre is made up of researchers in computer science, international relations, law, psychology, management, design, digital humanities, public policy, political Science, criminology, and sociology from five British universities including the University of Bristol.

REPHRAIN should be understood in the context of the UK government's *Online Harms White Paper* public consultation beginning in April 2019 [6], which sets out plans for new online safety measures; REPHRAIN's missions and outcomes are aligned with this paper [7].

REPHRAIN will focus on three core missions [8]:

1. Information collection
   (a) Surveillance
   (b) Interrogation
2. Information processing
   (a) Aggregation
   (b) Identification
   (c) Insecurity
   (d) Secondary Use
   (e) Exclusion
3. Information dissemination
   (a) Breach of Confidentiality
   (b) Disclosure
   (c) Exposure
   (d) Increased Accessibility
   (e) Blackmail
   (f) Appropriation
   (g) Distortion
4. Invasion
   (a) Intrusion
   (b) Decisional Interference

Figure 1.1: A Taxonomy of Privacy Violations [1].

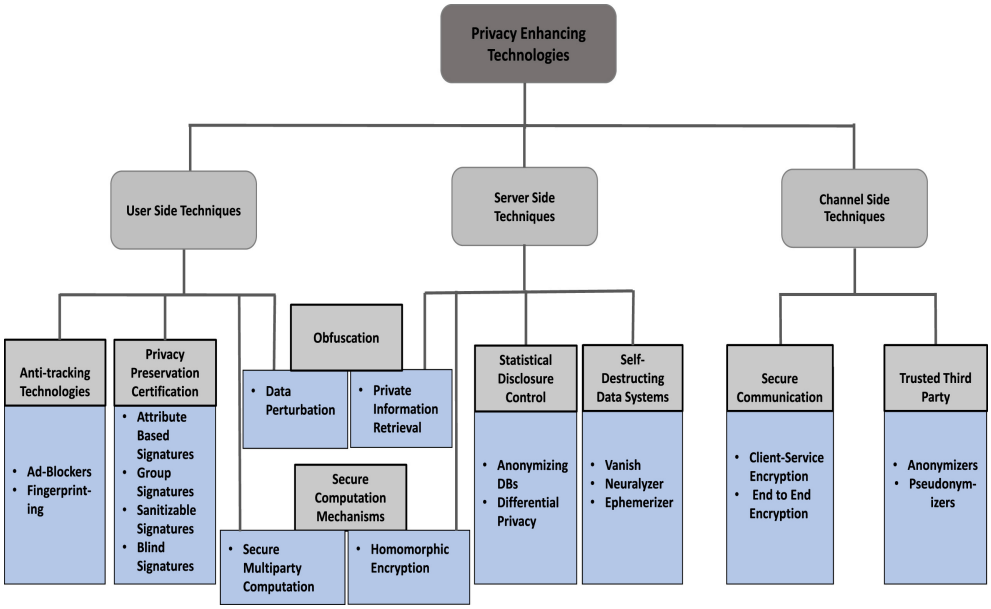


Figure 1.2: A Taxonomy of privacy enhancing technologies [3].

1. Delivering privacy at scale while mitigating its misuse to inflict harms.

2. Minimising harms while maximising benefits from a sharing-driven digital economy.

3. Balancing individual agency vs. the social good.

The three missions will require looking at Privacy Enhancing Technologies (PETs); including their capabilities, applications of PETs in addressing existing online harms, mitigating the potential abuse of PETs, embedding the PETs into infrastructures, and developing new PETs. In order to facilitate this REPHRAIN intends to build a toolbox of resources including a PETs testbed. The testbed will be used by researchers in developing, testing, and evaluating the PETs.

The finished REPHRAIN PETs testbed will likely include the following features:

- Support for testing various architectures and network topologies, including client-server and peer-to-peer applications; in order to accommodate testing a wide variety of different PETs.

- The ability to collect a variety of information such as packet captures and memory dumps for use in evaluating the privacy and safety properties of the application.

- Support different operating systems and platforms such as desktop and mobile apps. Including applications that compiled from available source code, or using pre-built binaries.

- Implement a high degree of automation, such that working with large test environments becomes feasible.

1.4 High Level Objectives

The overall aim of this project is to contribute to the REPHRAIN project by developing a prototype testbed for privacy enhancing technologies (PETs).

The high level objectives for the prototype cover a subset of the goals for the final REPHRAIN testbed, including to:

1. Implement support for testing client-server PET applications; spanning multiple networked devices.

2. Demonstrate the ability to capture network traffic.

3. Demonstrate how the testbed can be used in testing both desktop and mobile apps.

4. Provide a high level of automation, allowing setups to be easily and programmatically be deployed and replicated.

1.5 Existing Testbeds

There has been some existing research by Tekeoglu and Tosun [9] who have developed a privacy testbed for Internet-of-Things (IoT) devices. Their approach has some similar goals to this project in that it looks at capturing layer 2 and 3 network traffic. They note that the testbed enables experiments such as port vulnerability scans, checking what cipher suites are used (or not), and generally monitoring network traffic to see what data is being collected. However their testbed is different in that it is only designed for IoT devices; rather than general purpose PET applications.

1.6 Comparison to Similar Products

Chapter 2

Technical Background

In this chapter I will discuss some of the technologies which this project depends or builds upon.

2.1 Virtualization

> 'Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs). Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware.' [10]

Virtualization[1] will therefore be a very useful technology for the testbed, since it will allow us to model an environment consisting of multiple computers such as application clients and servers, and run them all within a single machine. In addition, modern CPU extensions (such as *Intel VT* and *AMD-V*) provide hardware-assisted / accelerated virtualization support, allowing the virtual machines to have near native performance which will help in meeting the goal of minimal overhead for the testbed.

In order to use virtualization a *Hypervisor* is required, this is the software layer sits between the physical hardware and manages the virtual machines. A hypervisor may run directly on the physical machine in place of a conventional operating system - as a Type 1 or *bare-metal* hypervisor, or run within a separate host operating system - as a Type 2 or *hosted* hypervisor.

2.1.1 *KVM*

For the prototype testbed I will be using *Kernel-based Virtual Machine* (KVM), which is a kernel module for the Linux operating system that allows it to function as a hypervisor. The

[1] I am a British citizen, and this is a dissertation at a British university, and therefore I am very much aware that the correct spelling in British English is 'virtualisation', however the majority of platforms, libraries, and sources I will be referencing use the US English spelling, so I have chosen to do the same. Likewise the same applies for 'containerization'.

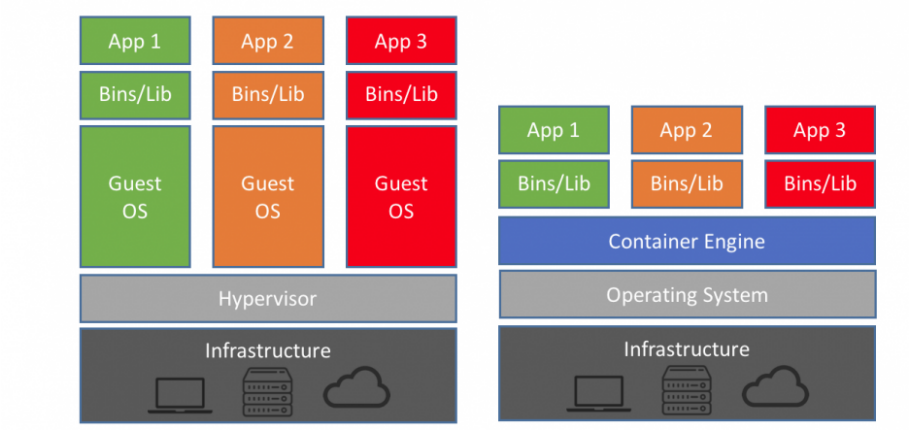


Figure 2.1: Platform Virtualization vs Containerization[3]

advantage is that *KVM* (and the Linux kernel itself) are free and open-source software under GNU licenses, and it is a stable and mature platform [11]. In userspace *QEMU*[2] may then use KVM to provide a full virtualization platform.

libvirt is an open-source toolkit for managing virtualization platforms [12], it supports *QEMU/KVM* as well as hypervisors from other vendors. It is written in C, with bindings available in many other programming languages, making it a suitable library for developing the testbed. Support for other platforms also means it would be easier to add support for additional hypervisors in future.

2.2 Containerization

Having discussed full platform / machine virtualization it is worth also mentioning containerization which is an alternative lightweight approach to virtualizing applications. In particular there is the Docker platform for Linux containers which has become very popular in recent years [13].

Unlike full platform virtualization, containerization does not virtualize a whole computer (or require hardware acceleration), instead it uses namespacing within the host operating system kernel to create an isolated environment. This has many practical uses, making deploying software and services very quick and easy, but unfortunately it is not ideal for our testbed, since it would mean all applications would have to use the same operating system; it couldn't be used to simulate different platforms.

While I will not be using containerization, the *Docker Compose* tool [14] used for orchestrating containers has provided some useful inspiration for the testbed. *Docker Compose* is a command line program with two core subcommands *up* and *down* which are used to either build or destroy a set of containers as defined in a YAML configuration file. The configuration file

[2]Note *QEMU* can also function as its own independent type 2 hypervisor but *KVM* is required to enable hardware acceleration, see https://www.packetflow.co.uk/what-is-the-difference-between-qemu-and-kvm/

[3]Graphic from https://blog.netapp.com/blogs/containers-vs-vms/

may define a list of containers, each with options including an image to download, an entry command to run, volumes to attach, and environment variables, the config may also define virtual networks and attach them to the containers. These are all very useful features in line with the goals for the testbed, and as such I will try to replicate them but within a fully virtualized environment (*QEMU/KVM*).

2.3 Virtual Networks

Once we have created virtual machines, a hypervisor needs the ability to bridge traffic between the virtual machines and the external network [15]. One method of doing of doing so with *QEMU/KVM* is to use the *Linux Bridge*[4], this is a virtual layer 2 switch built into the *Linux* kernel, that can be managed with the `bridge-utils` package (or equivalent).

There is also an alternative virtual networking solution for *Linux* (and other platforms) - *Open vSwitch*. The advantages of *Open vSwitch* over *Linux Bridge* include that it has easy management via SDN (explained below), it can support more complex network protocols, and it can function as a Layer 3 router (as opposed to just a Layer 2 bridge).

2.3.1 Software Defined Networking (SDN)

In order to understand Software Defined Networking (SDN) we must first explain how networking devices such as switches and routers function.

The *data plane* refers to the functions of a network device that actually process and forward packets between interfaces. For example the ASIC (application specific integrated circuit) logic that matches packets to a routing table.

The *control plane* refers to the processes that a network device uses in order to control the data plane; to determine which paths a particular packet should take. For example a dynamic routing protocol such as OSPF that programs a routing table.

The *management plane* refers to the protocols that a network administrator can use to manage and configure the network device. For example the SSH protocol can be used to remotely connect a console.

SDN refers to the practice of removing the control plane responsibilities of each individual network device, and centralising the control plane within an SDN controller [16, p. 760].

The benefit of SDN is that it can simplify network management since the network can be managed from a single controller. This is especially useful in this scenario of a virtual testbed because researchers are likely to want to be able to quickly conduct experiments varying the structure of the virtual network while testing PETs.

OpenFlow is a protocol that implements SDN by allowing remote access / control to the data plane of a switch or router [17]. *OpenFlow* is supported by *Open vSwitch*, and there are a

[4] https://wiki.linuxfoundation.org/networking/bridge

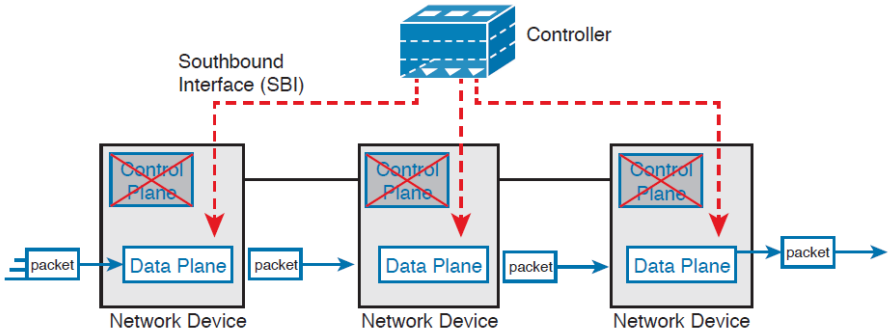


Figure 2.2: Use of an SDN controller [16, p. 767]

number of free and open-source SDN controllers that also support *OpenFlow* such as *Floodlight*[5] and *OpenDaylight* (ODL)[6].

2.4 The Rust Language

As you will see in Chapter 3, I have chosen to use the Rust programming language for developing the testbed. Although there are no doubt many languages which could have been used, I will provide some background on Rust, and its advantages for this project.

Rust is a modern systems programming language, originally developed by Mozilla, with its first stable release in 2015. It is designed with a focus on performance, safety and concurrency.

It has some key advantages:

- Excellent performance; on par with *C/C++*[7].
- Easy interaction with *C* libraries via FFI, making the *libvirt* [18] bindings possible.
- A simple to use package manager - *Cargo*, along with a rich ecosystem[8] of libraries / crates[9].
- Modern functional constructs such as sum types and pattern matching.
- Memory and thread safety are enforced at compile time [19].
- The compiler and standard library support a large number of platforms.

2.5 *cloud-init*

One of the goals of the project is to support a high level of automation when building a test environment. So far we have looked at technologies we can use to create virtual machines, but that would still leave the tester with a lot of work in terms of installing the operating systems

[5] https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview
[6] https://www.opendaylight.org/
[7] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
[8] https://crates.io/
[9] Rust packages managed through the *Cargo* package manager are known as *Crates*

and software on the virtual machines.

One approach the tester could take is pre-building a machine step by step, installing all their software and configuring it appropriately, and then saving a disk image, which they could attach to virtual test machines. However that approach has some weaknesses; first of all if they wish to change on of the earlier steps such as their choice of operating system it would require starting the process from scratch, another problem is that each clone of the disk image would be the same, and customisation would require manually logging in to each individual instance. Also if the tester wants to transfer or replicate the testbed it would mean copying a series of potentially large disk images with duplicate data.

The problem of automatically initialising virtual machines is not unique to this testbed project, in fact it is shared by public cloud providers (such as *Amazon AWS*, *Microsoft Azure*, *Oracle OCI*, etc.), who need to deploy various operating systems (of many different versions) and then configure them so the customer can remotely connect. As a result a standard system for initialising cloud machines has been developed called *cloud-init* [20] - which has support for many popular operating systems, cloud platforms and data sources.

It works by having a *cloud-init* package already installed as part of the operating system, in a 'cloud image', usually available directly from the operating system maintainers [10]. On the first boot of the image *cloud-init* will check for supported data sources, (such as from a list of supported URLs) from which to obtain user data and instance meta data. The data can include things such locale, hostname, and SSH keys which are then automatically applied to the instance.

[10] For example https://cloud-images.ubuntu.com/

Chapter 3

Project Execution

3.1 *kvm-compose*

To provide the core functionality of the testbed I have developed a command line tool called *kvm-compose*, using the technologies described in chapter 2, namely *QEMU/KVM*, *libvirt*, *Open vSwitch*, *cloud-init*, and the *Rust* language.

3.1.1 Dependencies and Installation

In order to compile the project it requires the *Rust* compiler[1], and the `libvirt-dev` and `libssl-dev` packages (or system equivalent). The *bash* script `kvm-compose/install.sh` may be used to compile *kvm-compose*, install the binary into a system folder, and change its ownership to *root*. Note that the *root* permission is added so that the utiltiy may access `qemu:///system` and modify the *Open vSwitch* and other networking configurations.

To successfully use *kvm-compose* the following packages should be installed (or equivalent): `qemu-kvm libvirt-daemon-system libvirt-clients openvswitch-switch`. Hardware acceleration for virtualization should be enabled in the system, and if required for the application being tested - nested virtualization for KVM should also be enabled[2].

3.1.2 Command Line Interface (CLI)

As mentioned in section 2.2 the CLI is partly inspired by *docker-compose*. It has two main subcommands `up` and `down` which create or destroy a testbed environment. It will look for a configuration file called `kvm-compose.yaml` in the current directory (unless otherwise specified with the `--input` flag). And the current directory name is taken to be used as the 'project name' (again unless specified with the `--project-name` flag). The *clap* crate was used to automatically parse the command line arguments in a type-safe manner.

[1] https://www.rust-lang.org/tools/install
[2] https://docs.fedoraproject.org/en-US/quick-docs/using-nested-virtualization-in-kvm/

```
kvm-compose 1.0
Jacob Halsey

USAGE:
kvm-compose [FLAGS] [OPTIONS] <SUBCOMMAND>

FLAGS:
-h, --help       Prints help information
--no-ask     Suppress (accept) continue prompts
-V, --version    Prints version information

OPTIONS:
--input <input>                  Configuration file [default: kvm-compose.yaml]
--project-name <project-name>    Defaults to the current folder name
-v, --verbosity <verbosity>

SUBCOMMANDS:
cloud-images    List supported cloud images
down            Destroy all virtual devices in the current configuration
help            Prints this message or the help of the given subcommand(s)
machine         Configure individual machine
up              Create all virtual devices in the current configuration
```

Figure 3.1: *kvm-compose* usage / help output

The basic principle for the `up` command is to first iterate over all the configured network bridges, if each bridge does not already exist it is then created as per the configuration, with the project name as a prefix. The prefix is used to reduce the possibility that resources in the current testbed project will conflict with other existing configuration on the system or other testbed projects. After creating the bridges it then iterates over all the configured virtual machines, if it does not exist it is created according to its specification (with the project prefix) and then launched. Files required for the virtual machines to function such as disk images are placed in the current directory. The `down` command does the opposite in reverse order; stopping and removing the machines (and any left over files), and then removing the bridges.

There is also the `machine` subcommand which can be used to `up` or `down` individual machines without affecting the rest of the testbed, in the event the user wishes to just make configuration changes to a small number of virtual machines.

3.1.3 Configuration File Format

The configuration file is expected to be in the YAML format [21], internally *kvm-compose* uses the *serde* and *serde-yaml* crates to parse the file. The root of the config file should look like:

```
machines:                       # List of testbed virtual machines
bridges:                        # List of testbed virtual bridges
ssh_public_key: ssh-rsa...      # Your SSH public key
```

An example machine config could look like:

```
- name: example1                # (VM will be prefixed with project name)
  memory_mb: 4096               # Optional: default 512MiB
  cpus: 4                       # Optional: default 1
  disk:                         # Two variants: cloud_image or existing_disk
    cloud_image:
      name: ubuntu_18_04
      expand_gigabytes: 25      # Optional: Adds additional space to the virtual disk
  interfaces:                   # List of connected network interfaces
    - bridge: br0               # (A bridge defined in the same project)
  run_script: ./script.sh       # Optional: path to a script
  context: ./file.txt           # Optional: path to a file or folder
  environment:                  # Dictionary of arbitary environment variables
    key: value                  # Use /etc/nocloud/env.sh *key* to query
```

Alternatively an existing disk image may be used instead of a cloud image:

```
    existing_disk:
      path: ./disk.img          # Path to the disk image
      driver_type: qcow2        # raw or qcow2 (Default: raw)
      device_type: disk         # disk or cdrom (Default: disk)
      readonly: false           # Default: false
```

Assuming that the disk image supports *cloud-init*[3] at first boot the following will happen:

- The machine `name` (with project prefix) is used as the hostname.
- The SSH public key is injected into the instance.
- File(s) specified in `context` are copied into the `/etc/nocloud/context` directory.
- The `run_script` is executed, with its output log saved into `/etc/nocloud/`.

An example virtual bridge config could look like:

```
- name: br0                     # (Bridge will be prefixed with project name)
      # Optionally connect to a physical interface on the host
      # Such as to allow the virtual machines internet access
  connect_external_interfaces: [eth0]
      # When 'stealing' one of the host's adapters
      # you will need to assign the new virtual adapter an IP
  enable_dhcp_client: true
      # Optional: Specify an SDN controller to use
```

[3] *cirros-init* is also supported but only when used with the *cirros* cloud image

```
  controller: tcp:127.0.0.1:6653
      # Optional: Specify which protocol to use for the SDN controller
  protocol: OpenFlow13
```

3.1.4 Interaction with *QEMU/KVM*

I have used the *libvirt* library for interacting with *QEMU/KVM*, via the bindings to access it from the *Rust* language: *libvirt-rust* [18]. One of the benefits of the *kvm-compose* tool is that it automates the creation of *libvirt Domain* XML configurations (with the help of the *xml-rs* crate). These are otherwise tedious to write, for example it generates the following XML to send to *libvirt*:

```
<?xml version="1.0" encoding="UTF-8"?>
<domain type="kvm">
  <name>signal-client1</name>
  <cpu mode="host-model" />
  <vcpu>2</vcpu>
  <memory unit="MiB">2048</memory>
  <os>
    <type arch="x86_64">hvm</type>
  </os>
  <devices>
    <graphics type="vnc" port="-1" autoport="yes" />
    <disk type="file" device="disk">
      <driver name="qemu" type="qcow2" />
      <source file="/home/jacob/independent-project/examples/signal/client1-cloud-disk.img"
      <target dev="hda" bus="ide" />
    </disk>
    <disk type="file" device="cdrom">
      <driver name="qemu" type="raw" />
      <source file="/home/jacob/independent-project/examples/signal/client1-cloud-init.iso"
      <readonly />
      <target dev="hdb" bus="ide" />
    </disk>
    <interface type="bridge">
      <source bridge="signal-br0" />
      <virtualport type="openvswitch" />
    </interface>
  </devices>
  <sysinfo type="smbios">
    <system>
      <entry name="serial">ds=nocloud;</entry>
```

```
    </system>
  </sysinfo>
</domain>
```

3.1.5 Modifications to *libvirt-rust*

During my early work developing the project I discovered that under certain build setups / toolchains the *libvirt-rust* crate failed to compile, due to it containing function declarations that did not actually exist in the *libvirt* library. This is described in full detail in my issue report: https://gitlab.com/libvirt/libvirt-rust/-/issues/1.

I submitted a fixed version of *libvirt-rust* with these invalid functions removed, but also with a test case that builds the bindings in a static library, therefore checking that all the symbols (such as functions) did actually exist in the underlying *libvirt* library. The tests are automatically run as part of the CI/CD pipeline so this sort of problem should be prevented in future. The merge request was approved by the project maintainers: https://gitlab.com/libvirt/libvirt-rust/-/merge_requests/14.

Whilst working with *libirt* I also discovered that I was receiving duplicate error messages printed to the console (via *stdout/stderr*), this was because by default *libvirt* has an error handler configured to print all errors, but at the same time the *libvirt-rust* binding functions also returned a `Result<_, virt::error::Error>` type (the idiomatic approach in Rust), which on failure I was also printing to the console via the logging framework. So I also added a `clear_error_func()` to the merge request that binds to `virSetErrorFunc` so that the default error handler can be disabled.

3.1.6 Cloud Images

The currently supported cloud images may be listed using the `cloud-images` subcommand of *kvm-compose*. When a cloud image is chosen and launched *kvm-compose* takes the following steps:

1. Checks if the image has already been downloaded, the `directories` crate is used to resolve the path (`$HOME/.kvm-compose/` on Linux).
2. If not it will prompt the user before downloading; the download may be large so it makes sense to check first (this behaviour can be disabled with the `--no-ask` option for use in an unattended script). `Reqwest` is used for the download along with the `indicatif` crate to display progress.
3. A copy of the image with the machine name will be made in the current directory.
4. The disk image will be expanded if the `expand_gigabytes` option is specified.
5. The disk is attached to virtual machine via the *libvirt Domain* XML.

3.1.7 *cloud-init*

Section 2.5 provides an introduction to *cloud-init*, but I will now explain how it has been used in *kvm-compose*. When a virtual machine is created the virtual bios is configured to use the string `ds=nocloud` in the serial number. This indicates to *cloud-init* that the `NoCloud` datasource[4] is being used. On startup the *cloud-init* agent will then look for an *ISO9660/Joliet* or *vfat* filesystem with a volume label of `cidata`.

When creating virtal machines *kvm-compose* calls the `genisoimage` program[5] to generate a *Joliet* filesystem containing a *cloud-init* user data and instance metadata file. The instance metadata is a JSON encoded file; on `NoCloud` this can contain an automatically configured hostname, as well as any other keys. The keys also accessible via the `cloud-init query ds.meta_data.*key*` command inside the VM. The user data is a `#cloud-config` file in the YAML format, which when loaded by the *cloud-init* agent it is put through the *jinja* templating engine, allowing variable substitution from the instance metadata values[6]. The context file(s) are made available to the virtual machine as a *tarball* in the same *Joliet* volume. The `#cloud-config` file is used to list a number of startup commands necessary to provide the features needed by *kvm-compose* - such as launching and logging the `run_script` and extracting in the `context` file(s) to the local disk.

kvm-compose also supports the *Cirros* cloud image - this is a minimal Linux distribution by the *OpenStack Project*[7]. I have added support because due to its small footprint it could be useful when launching a large number of virtual machines on a resource constrained system. Note that when using *Cirros* the *cloud-init* setup works slightly differently because *Cirros* does not have a full *cloud-init* implementation - instead it uses *cirros-init* which doesn't support `#cloud-config`; the user data can only be a simple script[8]. Instance metadata keys are instead accessed via `cirros-query get *key*` command.

3.2 Examples

I have provided two example projects demonstrating how *kvm-compose* can be used to automatically create a virtual testbed environment for privacy enhancing technologies (PETs).

3.2.1 *DP3T*

Decentralized Privacy-Preserving Proximity Tracing (*DP3T*) is an open protocol developed for digital contact tracing during the COVID-19 pandemic, originating from the École polytechnique fédérale de Lausanne (EPFL) and ETH Zurich, it was developed with the intention of being less centralised than alternative solutions, and therefore more private [22]. Switzerland's

[4] https://cloudinit.readthedocs.io/en/latest/topics/datasources/nocloud.html
[5] https://linux.die.net/man/1/genisoimage
[6] See `/kvm-compose/assets/cloud_init.yaml`, the file is embedded in the binary using the `rust-embed` crate.
[7] https://docs.openstack.org/image-guide/obtain-images.html#cirros-test
[8] See `/kvm-compose/assets/cirros_init.sh`

national contact tracing app *SwissCovid* has been built using the *DP3T* protocol [23].

The files in `/examples/dp3t` demonstrate how *kvm-compose* can be used to build a test environment. First of all a virtual machine for the backend service is defined, which has a `run_script` that on launch:

1. Installs dependencies *Git* and *Maven*.
2. Clones and builds the `dp3t-sdk-backend`[9].
3. Installs and runs *dp3t* as a service.
4. Configures the network address of the server[10].

Android App

The *DP3T* project provides an SDK (Software Development Kit) for both *Android* and *Apple iOS* which is used to communicate with the backend server, it is this library which is used in the official implementations such as *SwissCovid*. The SDK also has a demo `calibration-app` (that uses the SDK library) with a simple user interface of diagnostic tools and info, for use in testing the system[11]. The `/examples/dp3t` contains a small patch to the source of the `calibration-app` that changes the server URL to the backend in the testbed (as well as setting the correct keys).

To run the `calibration-app` I have provided two different example machines. The first uses *Anbox* which is a compatibility layer for running *Android* apps within a *Linux* container[12], and the second uses the *Android Emulator* (from the official *Android SDK* / *Android Studio*)[13] which uses full virtualization instead[14]. For each of the two machines their `run_script` installs the dependencies and simulators. The `calibration-app` is made available in the machine in the `/etc/nocloud/context` folder, ready to be launched and used by the tester. I chose to provide two *Android* examples because each of the emulators have different strengths and weaknesses; the *Android Emulator* has more support for customising sensor input such as GPS and Gyroscope data, but it has poorer performance due to the extra layer of virtualization when compared to *Anbox*.

3.2.2 *Signal*

Signal is a messaging service built using its own custom end-to-end encryption protocol (the *Signal Protocol*), available for a number of platforms on mobile and desktop, designed with a

[9] https://github.com/DP-3T/dp3t-sdk-backend
[10] Unfortunately *Anbox* (discussed below) does not support using hostnames / DNS from the host machine's network, so a static address has been used.
[11] https://github.com/DP-3T/dp3t-sdk-android/tree/master/calibration-app
[12] https://anbox.io/
[13] https://developer.android.com/studio/run/emulator
[14] It therefore requires nested virtualization to be enabled, see section 3.1.1.

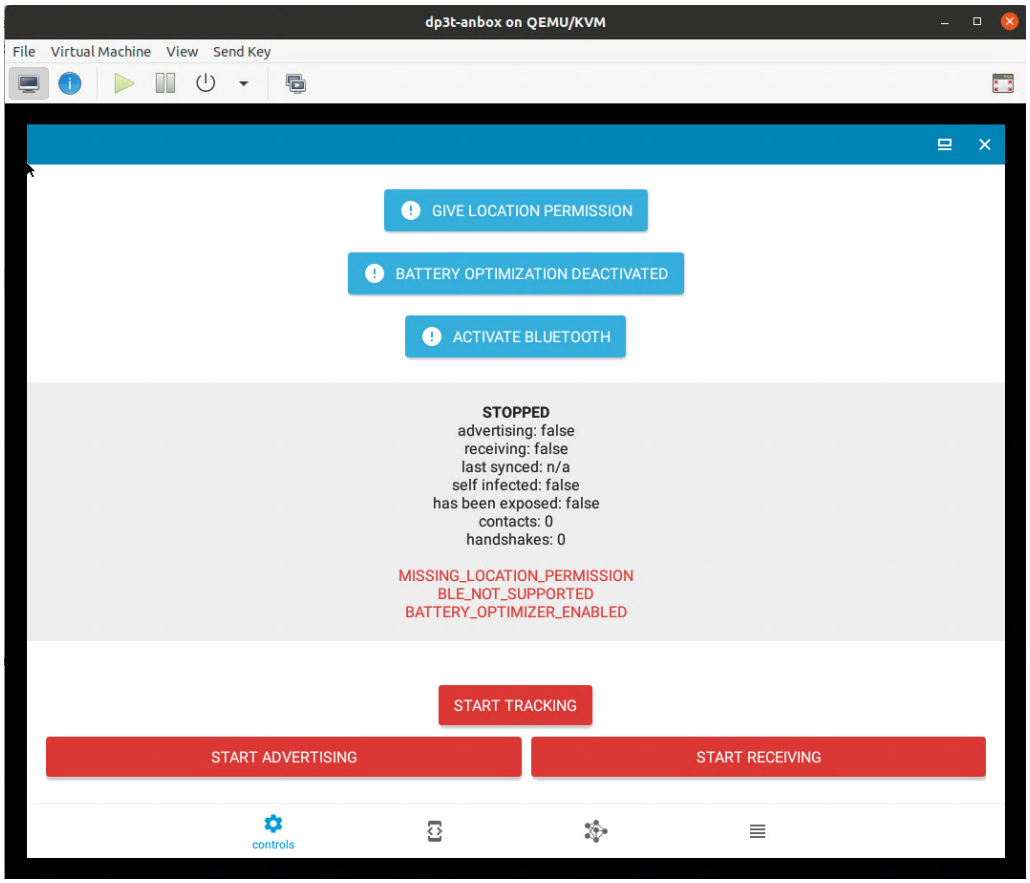


Figure 3.2: *DP3T* `calibration-app` running in *Anbox* on the testbed

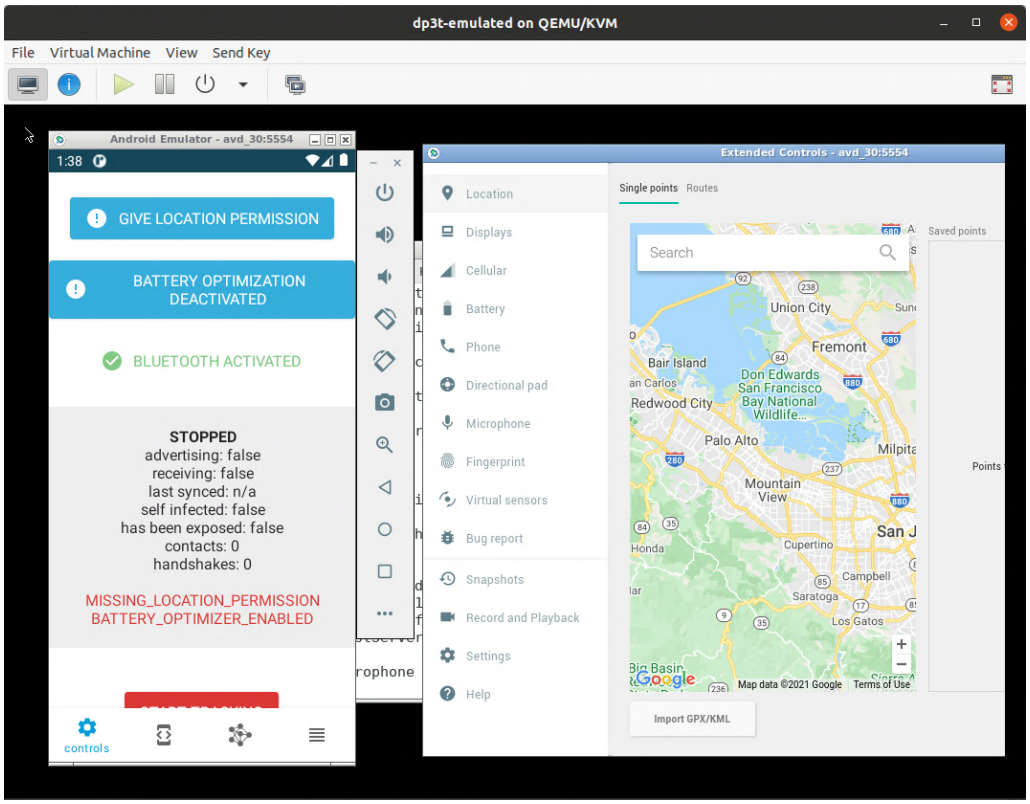


Figure 3.3: *DP3T* `calibration-app` running in *Android Emulator* on the testbed

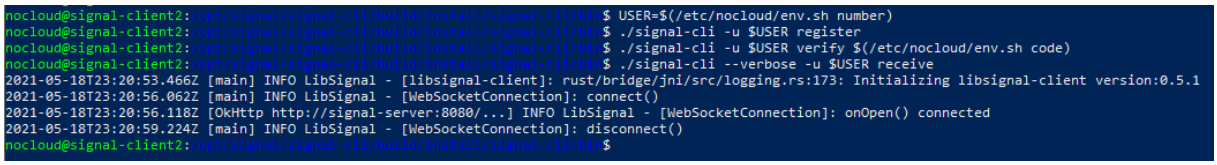


Figure 3.4: `signal-cli` registering and connecting via WebSockets to the testbed server

focus on privacy [24]. The standard service uses a central server hosted and maintained by the *Signal Foundation*, however the server code is also available under an open source license[15].

The files in `/examples/signal` demonstrate how *kvm-compose* can be used to build a test environment. First of all a virtual machine for the signal server is defined, which has a `run_script` that on launch:

1. Installs dependencies *Git*, *Maven*, *Docker* etc.
2. Clones and builds the *Signal* server.
3. Launches a number of dependent services (*Redis*, *Postgres*, *Nginx*) in *Docker* containers.
4. Initialises the databases.
5. Installs and runs the *Signal* server as a service; configuration files are provided via *kvm-compose*'s `context` option.

The Signal server's configuration has test device numbers defined, allowing them to be registered without requiring phone number verification. The *kvm-compose* configuration file then defines client machines using a `run_script` that:

1. Installs dependencies *Git* and *Java*.
2. Clones and builds `signal-cli`[16] which is an open source command line wrapper around the *Signal* SDK / client libraries.
3. Before building, applies a small patch to change the server address to the hostname of the *Signal* server within the testbed.

When the client's *Signal* server URL is patched, an `okhttp3.ConnectionSpec` is also provided allowing cleartext (non HTTPS) communication between the client and server. This is useful because it allows inspection of the packets which would otherwise be TLS encrypted. Unfortunately this exposed a bug in `libsignal-service-java` (a dependency of `signal-cli`) where the given `ConnectionSpec` is only respected for RESTful but not WebSocket communications with the *Signal* server. I have submitted a pull request to fix this: https://github.com/Turasa/libsignal-service-java/pull/28 which has since been merged.

[15] https://github.com/signalapp/Signal-Server
[16] https://github.com/AsamK/signal-cli

3.2.3 Capturing Traffic

When a test environment has been built (using the `kvm-compose up` command), network traffic can then be collected using the `ovs-tcpdump` utlity of *Open vSwitch*[17]. For example:

```
sudo ovs-tcpdump --span -i dp3t-br0 -w output.pcap
```

The `-i` option is where the bridge name is specified, in the format of the project name and bridge name (from the `kvm-compose.yaml` file) joined with a hyphen. What this does is create a temporary mirror port on the specified bridge, with the `--span` option indicating that traffic from all bridge ports should be mirrored.

Any additional options are passed onto the `tcpdump` application itself. These options could include various packet filters[18], in the example above the `-w` option is used to write the output as a packet capture file which may then be opened and analysed with software such as *Wireshark*[19].

3.3 Development Practices

During development I used the *Git* version control system, with free hosting from *GitHub Inc*, a `.gitignore` file was used to prevent temporary and user specific files being added to the repository. The *IntelliJ Rust* plugin for *CLion* proved very useful for code completion and other typical IDE features[20]. I also made use of the *GitHub Actions* CI/CD platform, with a workflow configured to check that the project builds and passes style checks on each push to the repository[21].

[17] https://docs.openvswitch.org/en/latest/ref/ovs-tcpdump.8/
[18] https://www.tcpdump.org/manpages/tcpdump.1.html
[19] https://www.wireshark.org/
[20] https://www.jetbrains.com/rust/
[21] See `.github/workflows/rust.yml`

Chapter 4

Critical Evaluation

4.1 Results of Deploying the Testbed Example Projects

For the *DP3T* example I was able to quickly build a testbed by using the `kvm-compose up` command. After waiting a few minutes for the machines to become ready I was able to login and use *Anbox* to run the `calibration-app`. Figures 4.1 and 4.2 show an infection report being sent using the app, and the resulting data being captured.

However one issue I found was that I had to manually inspect the output logs of the `run_script` to confirm that the virtual machines were setup. For example on an earlier attempt at running the *DP3T* testbed, the installation of the backend server failed because it was trying to download a dependency from *Bintray* which has just been discontinued, but I was not notified of this and had to manually inspect the virtual machine's logs to discover this. Fortunately I was able to apply a small patch to remove the *Bintray* dependency and get the installation working again[1].

Perhaps an improvement to the testbed could involve some sort of system so that the virtual machines can report their installation status directly back to the user, and alert them when it is ready for use, or otherwise if any errors have occurred.

Likewise I was able to do the same with the *Signal* example project, building a testbed by using the `kvm-compose up` command. Figure 4.3 shows some of the packet capture data as a result of using the `signal-cli` program within the testbed.

4.2 Testbed Failure Cases

Unlike other high level languages *Rust* does not implement exceptions, instead fallible functions should return a `Result<_, _>` type, where the left type is for success and the right type represents an error. Within *kvm-compose* I have used the `anyhow` crate to simplify error handling, since it

[1] The root cause of this is that the *DP3T* version is out of date - explained in 4.4.2

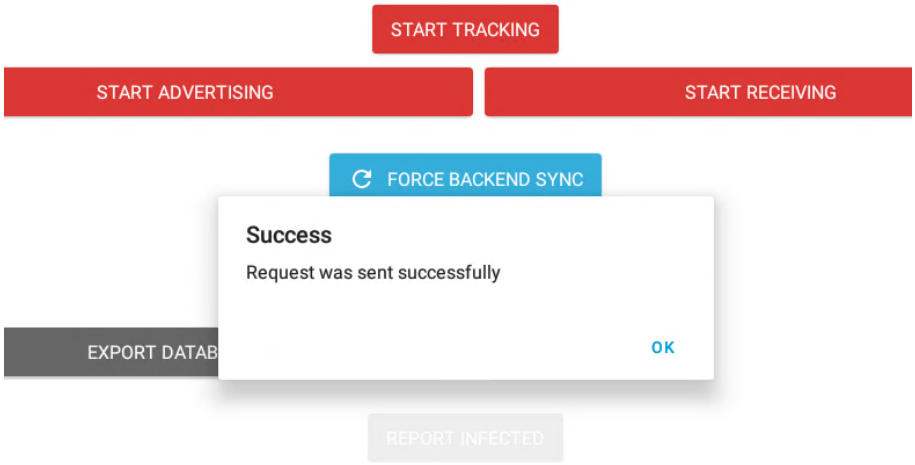


Figure 4.1: `calibration-app` successfully sending an infection report

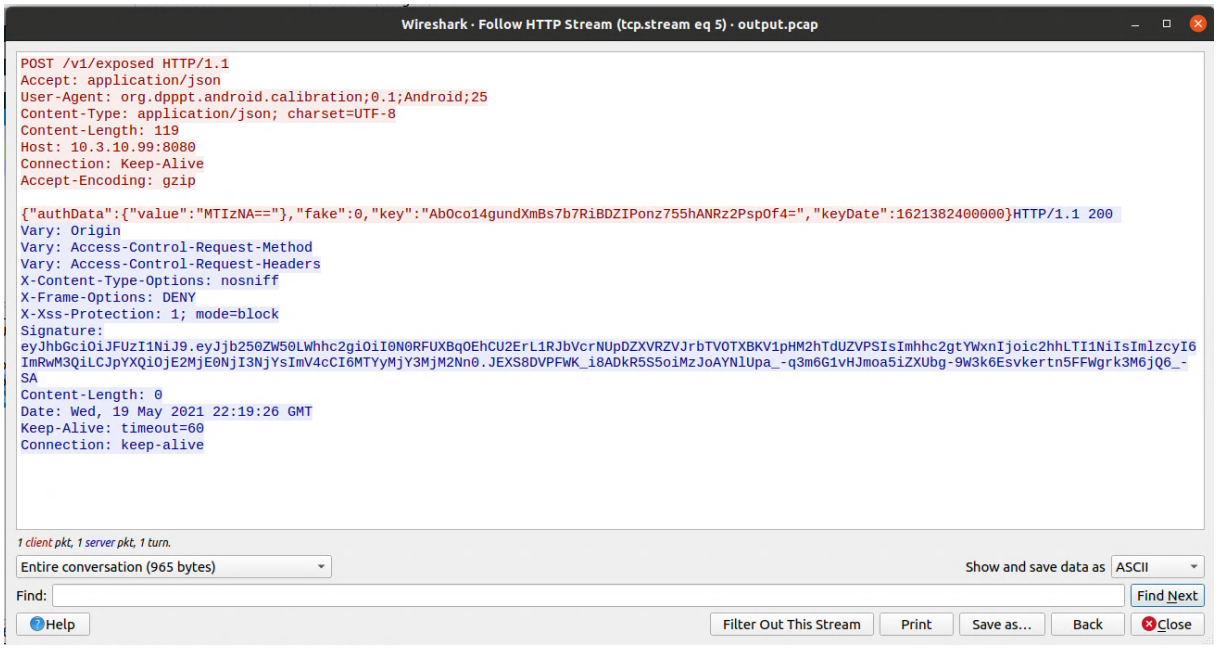
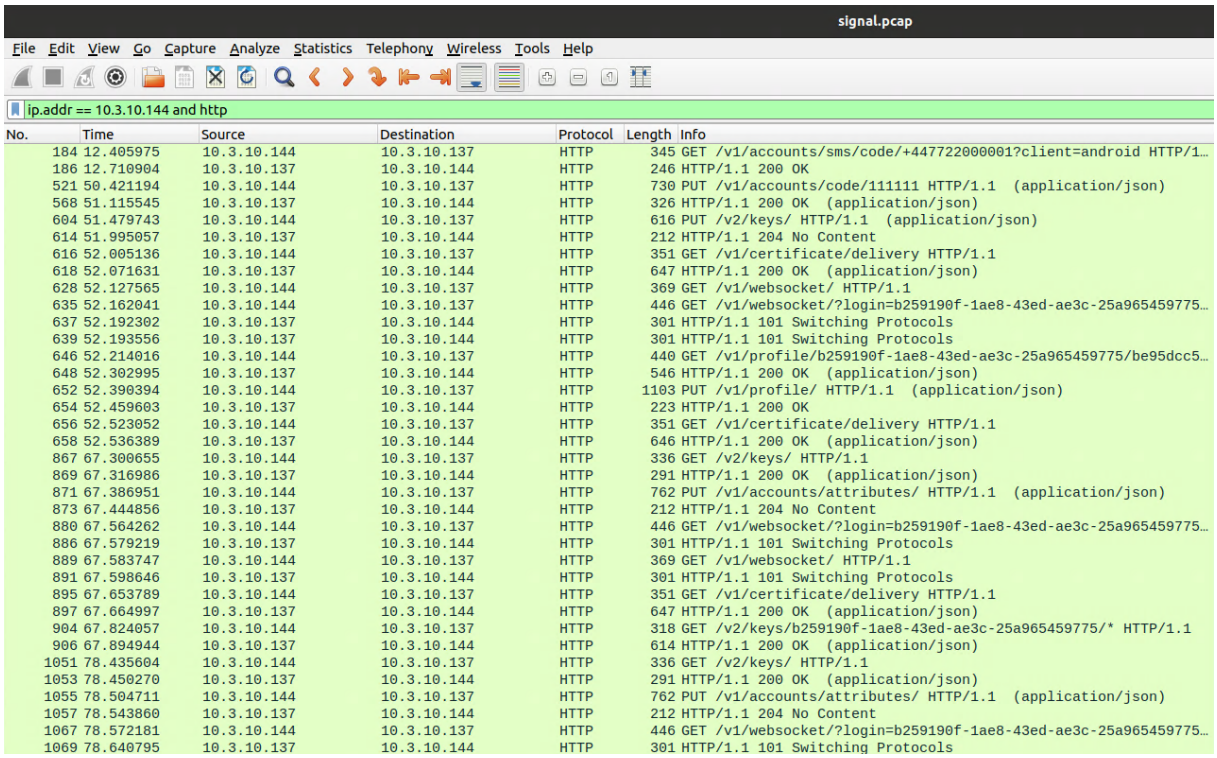


Figure 4.2: Packet capture of the send infection report



No.	Time	Source	Destination	Protocol	Length	Info
184	12.405975	10.3.10.144	10.3.10.137	HTTP	345	GET /v1/accounts/sms/code/+447722000001?client=android HTTP/1…
186	12.710904	10.3.10.137	10.3.10.144	HTTP	246	HTTP/1.1 200 OK
521	50.421194	10.3.10.144	10.3.10.137	HTTP	730	PUT /v1/accounts/code/111111 HTTP/1.1 (application/json)
568	51.115545	10.3.10.137	10.3.10.144	HTTP	326	HTTP/1.1 200 OK (application/json)
604	51.479743	10.3.10.144	10.3.10.137	HTTP	616	PUT /v2/keys/ HTTP/1.1 (application/json)
614	51.995057	10.3.10.137	10.3.10.144	HTTP	212	HTTP/1.1 204 No Content
616	52.005136	10.3.10.144	10.3.10.137	HTTP	351	GET /v1/certificate/delivery HTTP/1.1
618	52.071631	10.3.10.137	10.3.10.144	HTTP	647	HTTP/1.1 200 OK (application/json)
628	52.127565	10.3.10.144	10.3.10.137	HTTP	369	GET /v1/websocket/ HTTP/1.1
635	52.162041	10.3.10.144	10.3.10.137	HTTP	446	GET /v1/websocket/?login=b259190f-1ae8-43ed-ae3c-25a965459775…
637	52.192302	10.3.10.137	10.3.10.144	HTTP	301	HTTP/1.1 101 Switching Protocols
639	52.193556	10.3.10.137	10.3.10.144	HTTP	301	HTTP/1.1 101 Switching Protocols
646	52.214016	10.3.10.144	10.3.10.137	HTTP	440	GET /v1/profile/b259190f-1ae8-43ed-ae3c-25a965459775/be95dcc5…
648	52.302995	10.3.10.137	10.3.10.144	HTTP	546	HTTP/1.1 200 OK (application/json)
652	52.390394	10.3.10.144	10.3.10.137	HTTP	1103	PUT /v1/profile/ HTTP/1.1 (application/json)
654	52.459603	10.3.10.137	10.3.10.144	HTTP	223	HTTP/1.1 200 OK
656	52.523052	10.3.10.144	10.3.10.137	HTTP	351	GET /v1/certificate/delivery HTTP/1.1
658	52.536389	10.3.10.137	10.3.10.144	HTTP	646	HTTP/1.1 200 OK (application/json)
867	67.300655	10.3.10.144	10.3.10.137	HTTP	336	GET /v2/keys/ HTTP/1.1
869	67.316986	10.3.10.137	10.3.10.144	HTTP	291	HTTP/1.1 200 OK (application/json)
871	67.386951	10.3.10.144	10.3.10.137	HTTP	762	PUT /v1/accounts/attributes/ HTTP/1.1 (application/json)
873	67.444856	10.3.10.137	10.3.10.144	HTTP	212	HTTP/1.1 204 No Content
880	67.564262	10.3.10.144	10.3.10.137	HTTP	446	GET /v1/websocket/?login=b259190f-1ae8-43ed-ae3c-25a965459775…
886	67.579219	10.3.10.137	10.3.10.144	HTTP	301	HTTP/1.1 101 Switching Protocols
889	67.583747	10.3.10.144	10.3.10.137	HTTP	369	GET /v1/websocket/ HTTP/1.1
891	67.598646	10.3.10.137	10.3.10.144	HTTP	301	HTTP/1.1 101 Switching Protocols
895	67.653789	10.3.10.144	10.3.10.137	HTTP	351	GET /v1/certificate/delivery HTTP/1.1
897	67.664997	10.3.10.137	10.3.10.144	HTTP	647	HTTP/1.1 200 OK (application/json)
904	67.824057	10.3.10.144	10.3.10.137	HTTP	318	GET /v2/keys/b259190f-1ae8-43ed-ae3c-25a965459775/* HTTP/1.1
906	67.894944	10.3.10.137	10.3.10.144	HTTP	614	HTTP/1.1 200 OK (application/json)
1051	78.435604	10.3.10.144	10.3.10.137	HTTP	336	GET /v2/keys/ HTTP/1.1
1053	78.450270	10.3.10.137	10.3.10.144	HTTP	291	HTTP/1.1 200 OK (application/json)
1055	78.504711	10.3.10.144	10.3.10.137	HTTP	762	PUT /v1/accounts/attributes/ HTTP/1.1 (application/json)
1057	78.543860	10.3.10.137	10.3.10.144	HTTP	212	HTTP/1.1 204 No Content
1067	78.572181	10.3.10.144	10.3.10.137	HTTP	446	GET /v1/websocket/?login=b259190f-1ae8-43ed-ae3c-25a965459775…
1069	78.640795	10.3.10.137	10.3.10.144	HTTP	301	HTTP/1.1 101 Switching Protocols

Figure 4.3: Packet capture of `signal-cli` registration and messaging, viewed in Wireshark

reduces the need to define custom error types and has useful macros for manipulating errors [25].

Within *kvm-compose* there are many functions that could fail, these include file system operations, interactions with *QEMU/KVM* (via `libvirt-rust`), and calls to command line tools. All of these return `Result` types which are then either handled or passed upwards to the calling function. For example one of the functions I defined for configuring *Open vSwitch*:

```rust
pub fn add_br<T: AsRef<str>>(name: T) -> anyhow::Result<()> {
    let output = Command::new("sudo")
    .arg("ovs-vsctl")
    .arg("add-br")
    .arg(name.as_ref())
    .output()?;
    if !output.status.success() {
        let std_err = std::str::from_utf8(&output.stderr)?;
        bail!("{}", std_err);
    }
    Ok(())
}
```

If either the operating system fails to call the command, or the result of the command was not a success then the Rust function will return `Result::Err()` containing a description of what

went wrong. This approach is used throughout the program, so if there are any errors then *kvm-compose* will clean up and exit. The error description is printed to the console as part of the log.

The *kvm-compose* log output can be changed with the `--verbosity` flag, with levels of `[error, warn, info, trace]`. The higher levels could be useful to the user if they need to debug the program, for example the `trace` option will show the full XML output that is given to `libvirt`. Any errors or problems that occur when running *kvm-compose* are therefore presented very clearly to the user as soon as it is executed.

Errors that occur during the `run_script` stage of initialising the virtual machines are unfortunately less accessible. The `stdout` and `stderr` streams of script are logged and stored in `/etc/nocloud/`. But as noted in section 4.1 this will not appear to the user unless they manually look for those logs themselves. For example using SSH to connect to the virtual machine and running:

```
tail -f /etc/nocloud/run_script_out.txt
```

4.3 REPHRAIN Feedback

On the 4th May 2021 my co-supervisor presented my work on *kvm-compose* and the *DP3T* example using the slides and video (that I prepared for the poster day), as part of a REPHRAIN scoping session. My work received positive feedback.

One question that was asked focused on the performance and throughput of testbed. This is not something I have had time to conduct tests for, but I can theorise that it will largely depend on the performance of the system being used, since modern hardware acceleration means there will be minimal overhead from the virtualization.

4.4 Difficulties Creating Example Projects

During the creation of the example projects I discovered some limitations and issues that researchers may face when using the testbed.

4.4.1 *NHS* COVID App and Cloud Lock-In

Before creating the *DP3T* example project I was originally intending to test the UK's *NHS* COVID-19 contact tracing app implementation[2]. However it quickly became apparent that the service is highly coupled to *Amazon Web Services (AWS)*, and as such it is very difficult to run within the testbed. It is architectured using the serverless model, and depends on services includes using *AWS API Gateway*, *Lambda*, *DynamoDB* etc. Whilst it is indeed possible to run

[2] https://github.com/nihp-public/covid19-app-system-public

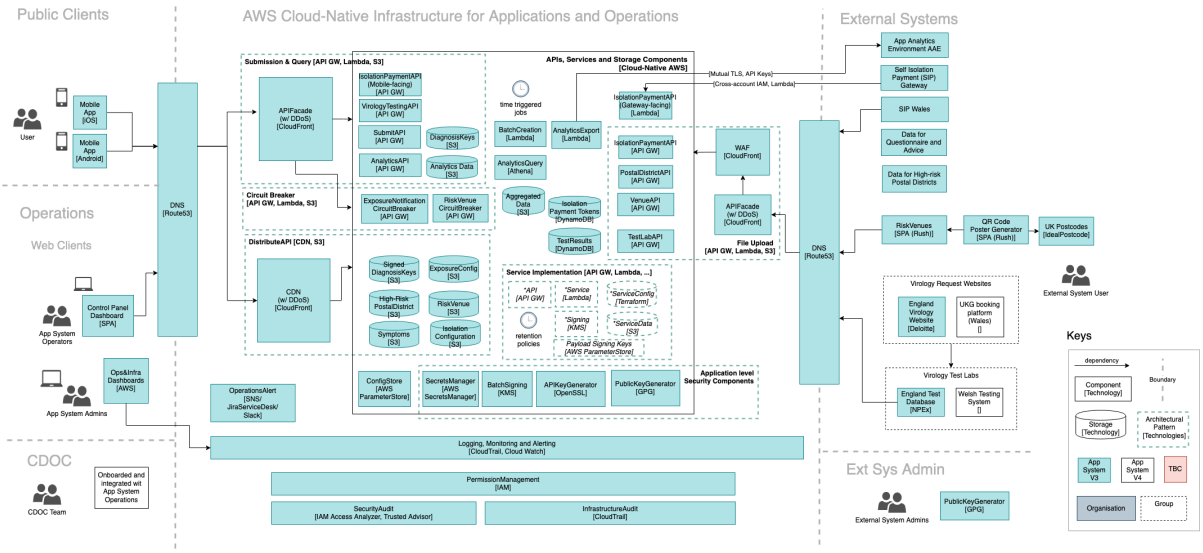


Figure 4.4: NHS COVID-19 App System Architecture

some of the *AWS* services locally[3], replicating the entire *AWS* infrastructure to simulate the COVID service would require a very substantial amount of time and effort; requiring code and configuration changes to enable connecting to local endpoints rather than the *AWS* regions.

This can be considered part of a broader problem of vendor lock-in in the context of modern cloud services; due to a lack of standardization and unique cloud service implementations, it means that applications are being developed that target a specific cloud service, and are not easy to move without incurring engineering costs [26]. This proves a challenge to the whole concept of a local virtual testbed in a world where technologies are being developed for specific proprietary cloud services, without support for self hosting. As such to test them locally would require code modifications, and the more the code is modified from the original, the less accurate and useful any test results can be considered, when compared to the real service.

4.4.2 Exposure Notifications API

After switching to the *DP3T* service I was able to setup the backend server within the testbed relatively easily, since it is implemented as a stand alone Spring Boot application[4]. However when attempting to deploy the mobile client I then discovered another issue: in order to use the *DP3T* `calibration-app` it is required to enable the debug mode for COVID-19 exposure notifications within the Android device settings, however that option was not available, since according to *Google*:

> To enable debug mode on a device, the primary account on the device must be a development account that is on the allowlist.[5]

[3] For example: https://hub.docker.com/r/amazon/dynamodb-local/
[4] https://spring.io/projects/spring-boot
[5] https://developers.google.com/android/exposure-notifications/debug-mode

Allowlist membership is only available to government health authorities, and official implementations of the app are distributed through *Google*'s *Play Store* and are digitally signed, hence allowing them to make use of the exposure notification API. But these are not useful for our testing purposes because we need to be able to change the URL to our own server. There does not currently appear to be any method of using the API for debugging or testing purposes[6], meaning it cannot be used within the testbed.

As a workaround for my example project so that I could demonstrate the *DP3T* service being used, I am using the `prestandard` edition, which used its own exposure detection protocol prior to the development of the Google one[7]. Whilst this does allows the app to launch successfully and data to be collected in the testbed it is no longer an accurate representation of the technology that is actually in live use. This highlights another similar issue with the testbed concept, in that despite being designed with a focus on privacy and security [27] some applications have been engineered without the ability to use in a test environment.

4.4.3 *Signal* Server

Setting up the *Signal* server created some challenges, which may again highlight the limitations of the testbed. The first difficulty is there does not appear to be any official documentation or guides to setup the server, and while there are some unofficial guides available[8], their use is limited due to substantial changes between different versions of *Signal*.

At first I attempted to use one of the latest versions (v5.80) of the *Signal* server however I immediately ran into difficulty because it has substantial dependencies on *Amazon Web Services (AWS)*. Version 5 has a `DynamicConfigurationManager` class that uses the `AmazonAppConfigClient` to connect to *AWS AppConfig*, using authentication from *Amazon*'s metadata service for *EC2* instances. The server will not start until it has successfully connected to *AppConfig*. Again this is another example of modern services being built tightly coupled to a cloud service provider, making it difficult to run in a test environment.

Nevertheless I tried to setup a *Signal* server in *AWS*, and was able to configure the *AppConfig* service, but I then discovered that it also depends on *AWS DynamoDB*. However due to the lack of documentation I was unable to successfully configure the schema of the *DynamoDB* tables. As a result I decided to use v4.97 of *Signal* server which does not have the *AWS* dependencies and as a result I was able to run it in the testbed. It did however take a substantial amount of time to work out how it should be configured, and in the process I have contributed to one of the guides (https://github.com/madeindra/signal-setup-guide/pull/83). The issue of complex but undocumented technologies could prove to be a problem for users of the testbed.

[6] https://github.com/google/exposure-notifications-android/issues/8

[7] https://github.com/DP-3T/dp3t-sdk-android#introduction

[8] See https://github.com/aqnouch/Signal-Setup-Guide and https://github.com/madeindra/signal-setup-guide

4.5 Have the Objectives Been Met?

In section 1.4 I outlined the high level objectives for the testbed, I will now discuss how well these objectives have been met.

The first objective requires support for running client-server applications in the testbed. My two example projects (*DP3T* and *Signal*) both use a client-server architecture and section 4.1 shows the applications communicating successfully. The virtual machines in those example projects were also able to communicate with the external network / internet in order to download their dependencies. The *kvm-compose* tool does in fact provide support for defining multiple network bridges, connecting multiple interfaces to each virtual machine, and attaching an SDN controller to the bridges, thereby providing a basis from which to start future work on deploying more complex networks and architectures.

The next objective looks at the ability to intercept and capture network traffic generated through the testbed. I consider objective to have been met, and the process has been documented in section 3.2.3. Captured network traffic from the example projects is shown in section 4.1. A potential improvement could be to integrate the packet capture commands as part of the *kvm-compose* utility to produce a more streamlined experience for the user.

The third objective requires being able to use two different types of applications; desktop and mobile apps. My *DP3T* example successfully demonstrates an *Android* app launching within the testbed, via two different methods. As mentioned in section 3.2.1 there are advantages and disadvantages to the two different methods (*Anbox* and *Android Emulator*). However a limitation experienced by both implementations (as well as other approaches to *Android* emulation) is that they do not support certain technologies such as Bluetooth or NFC[9]. Although Bluetooth is major part of its function, this does not necessarily have to be a complete barrier to testing *DP3T*; due to its open source SDK a tester could easily add code to produce simulated inputs in place of the Bluetooth communications, and then its client-server network communication can still be inspected as usual. Where this would however become a more severe limitation is in the context of closed source binaries, which cannot easily be engineered to use fake inputs.

The final objective requires that the testbed be highly automated and easy to replicate. My usage of *cloud-init* with *kvm-compose* (see section 3.1.7) has made it very easy to deploy command line applications, such as the servers for the *DP3T* and *Signal* examples. All it takes is a single command `kvm-compose up` within the project directory, and the servers are automatically downloaded, compiled, and installed. Such an environment would also be easy to replicate as it just requires copying a folder of small files, they can easily be maintained in a *Git* repository for example, and then running the `kvm-compose up` command would produce an identical setup. The level of automation is unfortunately less well developed when it comes to mobile apps, while

[9]For example see: https://developer.android.com/studio/run/emulator#limitations

in the *DP3T* example the emulators are installed automatically, it still requires the app itself to be launched and driven by a user using a window manager. It was also noted in 4.1 and 4.2 that the progress and status (success or failure) during the virtual machine `run_script` phase are not easily accessible, which also impedes full automation.

Chapter 5

Conclusion

5.1 Achievements

I have built a virtual testbed solution, capable of deploying virtual machines and virtual networks. The testbed can automatically configure the virtual machines with SSH keys, files, variables, and a boot script. I have demonstrated how the testbed can be used for deploying two different privacy enhancing technologies (PETs) including *DP3T* a COVID contact tracing app, and *Signal* an encrypted messaging platform. And how the testbed can be used for collecting network traffic data for research purposes. The testbed will be used as a prototype to inform the development of REPHRAIN's PET testbed, as part of a major national programme of research into online safety. During the development process I have also made contributions to three open source projects which have been accepted by their respective maintainers.

5.2 Project Status

5.3 Open Problems and Future Ideas

In section 4.4 I discovered that applications that applications which have been built to run in the cloud can be difficult to deploy in local environments outside of the commercial cloud infrastructure. With more and more modern applications being architectured as 'cloud-native" [28] this will be a problem the REPHRAIN testbed will have to consider and work around.

While my prototype demonstrates support for client-server applications, it has not yet been tested when simulating a more complex network environment. The *Open vSwitch (OVS)* bridges that I have used to provide the networking features are indeed capable of implementing layer 3 routing and more advanced topologies, but at present this would require manual configuration using the *OVS* commands. Future development of the PET testbed could look at providing easily accessible and automated support for deploying alternative and more complex topologies to client-server, such as peer-to-peer applications, and multi subnet networks. As well as providing suitable examples for how to use the features.

Bibliography

[1] D. J. Solove, "A taxonomy of privacy," *University of Pennsylvania Law Review*, vol. 154, no. 3, pp. 477–564, 2006. [Online]. Available: https://www.law.upenn.edu/journals/lawreview/articles/volume154/issue3/Solove154U.Pa.L.Rev.477%282006%29.pdf.

[2] D. Buckley, *Privacy enhancing technologies for trustworthy use of data*, Feb. 2021. [Online]. Available: https://cdei.blog.gov.uk/2021/02/09/privacy-enhancing-technologies-for-trustworthy-use-of-data/.

[3] N. Kaaniche, M. Laurent, and S. Belguith, "Privacy enhancing technologies for solving the privacy-personalization paradox: Taxonomy and survey," *Journal of Network and Computer Applications*, vol. 171, p. 102 807, 2020, ISSN: 1084-8045. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804520302794.

[4] UKRI, *Who we are*, Nov. 2020. [Online]. Available: https://www.ukri.org/about-us/who-we-are/.

[5] ——, *New centre launched to keep citizens safe online*, Oct. 2020. [Online]. Available: https://www.ukri.org/news/new-centre-launched-to-keep-citizens-safe-online/.

[6] Department for Digital, Culture, Media and Sport, *Online harms white paper*, Dec. 2020. [Online]. Available: https://www.gov.uk/government/consultations/online-harms-white-paper.

[7] REPHRAIN, *Online harms*. [Online]. Available: https://www.rephrain.ac.uk/online-harms/.

[8] ——, *Missions*. [Online]. Available: https://www.rephrain.ac.uk/missions/.

[9] A. Tekeoglu and A. S. Tosun, "A testbed for security and privacy analysis of iot devices," in *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, IEEE, 2016, pp. 343–348. [Online]. Available: https://ieeexplore.ieee.org/document/7815045.

[10] IBM Cloud Education, *Virtualization: A complete guide*, Jun. 2019. [Online]. Available: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide.

[11] Red Hat, *Kvm vs. vmware*. [Online]. Available: https://www.redhat.com/en/topics/virtualization/kvm-vs-vmware-comparison.

[12] libvirt, *The virtualization api*. [Online]. Available: https://libvirt.org/index.html.

[13] S. J. Vaughan-Nichols, *What is docker and why is it so darn popular?* Mar. 2018. [Online]. Available: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/.

[14] Docker Inc, *Overview of docker compose*, 2021. [Online]. Available: https://docs.docker.com/compose/.

[15] Open vSwitch, *Why open vswitch?* [Online]. Available: https://github.com/openvswitch/ovs/blob/master/Documentation/intro/why-ovs.rst.

[16] W. Odom, *CCNA Routing and Switching ICND2 200-105 Official Cert Guide, Academic Edition*, ser. Official Cert Guide. Pearson Education, 2016, ISBN: 9780134514086.

[17] N. McKeown *et al.*, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. [Online]. Available: https://doi.org/10.1145/1355734.1355746.

[18] libvirt, *Libvirt-rust*. [Online]. Available: https://gitlab.com/libvirt/libvirt-rust.

[19] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari´ c, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17, Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 156–161, ISBN: 9781450350686. DOI: 10.1145/3102980.3103006. [Online]. Available: https://doi.org/10.1145/3102980.3103006.

[20] Canonical, *Cloud-init - the standard for customising cloud instances*. [Online]. Available: https://cloud-init.io/.

[21] yaml.org, *The official yaml web site*. [Online]. Available: https://yaml.org/.

[22] D. Busvine, *Rift opens over european coronavirus contact tracing apps*, Apr. 2020. [Online]. Available: https://www.reuters.com/article/uk-health-coronavirus-europe-tech-idUKKBN2221U6?edition-redirect=uk.

[23] Swissinfo, *Contact tracing app ready this month, says expert*, May 2020. [Online]. Available: https://www.swissinfo.ch/eng/sci-tech/digital-solution_contact-tracing-app-could-be-launched-in-switzerland-within-weeks/45706296.

[24] Signal Foundation, *Speak freely*. [Online]. Available: https://signal.org/en/.

[25] N. Groenen, *Rust: Structuring and handling errors in 2020*, May 2020. [Online]. Available: https://nick.groenen.me/posts/rust-error-handling/.

[26] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective," *Journal of Cloud Computing*, vol. 5, no. 1, pp. 1–18, 2016. [Online]. Available: https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-016-0054-z.

[27] Google and Apple, *Exposure notifications: Helping fight covid-19*. [Online]. Available: https://www.google.com/covid19/exposurenotifications/.

[28] A. Patrizio, *What is cloud-native? the modern way to develop applications*, Jun. 2018. [Online]. Available: https://www.infoworld.com/article/3281046/what-is-cloud-native-the-modern-way-to-develop-software.html.