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.

Sunday 16th 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, Sunday 16th 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 the *libvirt-rust* language bindings open source library.
- 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

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 [1]. In October 2020 the UKRI announced the creation of the National Research Centre on Privacy, Harm Reduction and Adversarial Influence Online (REPHRAIN) [2]. 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 [3], which sets out plans for new online safety measures; REPHRAIN's missions and outcomes are aligned with this paper [4].

REPHRAIN will focus on three core missions [5]:

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 researchers in developing, testing, and evaluating the PETs. The aim of this project is to develop a prototype for this testbed.

1.1 What are Privacy Enhancing Technologies (PETs)?

Before we discuss Privacy Enhancing Technologies we must consider what we mean by "privacy". REPHRAIN is primarily using the definitions set out by D. J. Solove in his 2006 article *A Taxonomy of Privacy* [6, 4]. Solove notes that the definition of privacy has often been very

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 [6].

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 [7]. 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. [8] have defined a more comprehensive classification of PETs (see Figure 1.2).

1.2 Existing Solutions

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.

1.3 High Level Objectives

Overall the high-level objective of this project is to develop a simple and lightweight testbed platform for evaluating PETs:

- The testbed should support testing various architectures and network topologies, including client/server and peer-to-peer applications, to accommodate a variety of PETs.

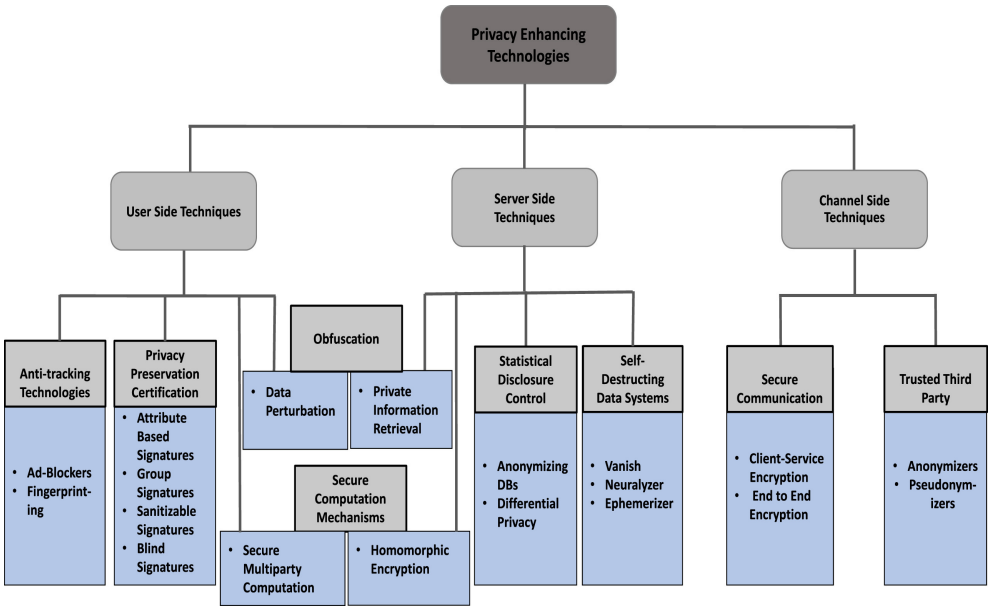

Figure 1.2: A Taxonomy of privacy enhancing technologies [8].

- The testbed must be able to collect information such as packet captures for use in evaluating the privacy properties.
- The testbed should support different platforms such as desktop and mobile apps, and both applications where the source code is available or only pre-built binaries.
- The testbed should enable a high level of automation, such that working with large test environments because feasible, and the setup can easily and programmatically be replicated.

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.

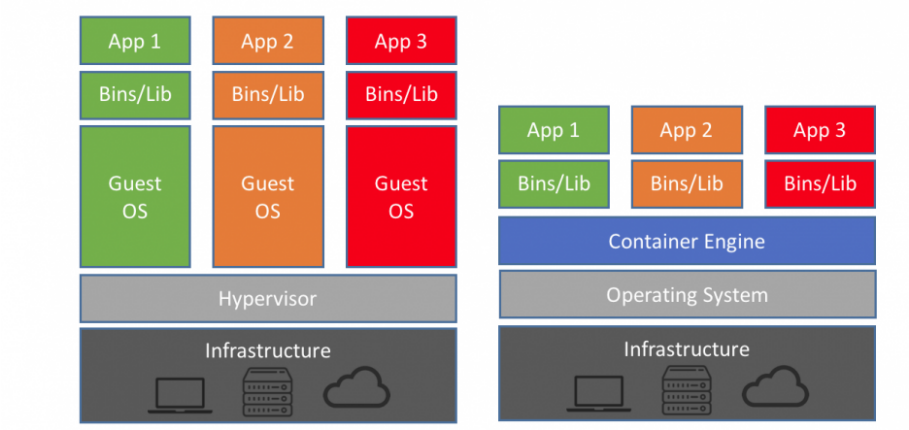

Figure 2.1: Platform Virtualization vs Containerization[3]

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

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 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

[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*

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
```

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

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

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()` that binds to `virSetErrorFunc` along with my changes, 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 App

3.2.2 Signal

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

[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`

other typical IDE features[9]. 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[10].

[9] https://www.jetbrains.com/rust/
[10] See `.github/workflows/rust.yml`

Chapter 4

Critical Evaluation

A topic-specific chapter, of roughly 15 pages

This chapter is intended to evaluate what you did. The content is highly topic-specific, but for many projects will have flavours of the following:

1. functional testing, including analysis and explanation of failure cases,
2. behavioural testing, often including analysis of any results that draw some form of conclusion wrt. the aims and objectives, and
3. evaluation of options and decisions within the project, and/or a comparison with alternatives.

This chapter often acts to differentiate project quality: even if the work completed is of a high technical quality, critical yet objective evaluation and comparison of the outcomes is crucial. In essence, the reader wants to learn something, so the worst examples amount to simple statements of fact (e.g., "graph X shows the result is Y"); the best examples are analytical and exploratory (e.g., "graph X shows the result is Y, which means Z; this contradicts [1], which may be because I use a different assumption"). As such, both positive *and* negative outcomes are valid *if* presented in a suitable manner.

Chapter 5

Conclusion

A compulsory chapter, of roughly 5 pages

The concluding chapter of a dissertation is often underutilised because it is too often left too close to the deadline: it is important to allocation enough attention. Ideally, the chapter will consist of three parts:

1. (Re)summarise the main contributions and achievements, in essence summing up the content.

2. Clearly state the current project status (e.g., "X is working, Y is not") and evaluate what has been achieved with respect to the initial aims and objectives (e.g., "I completed aim X outlined previously, the evidence for this is within Chapter Y"). There is no problem including aims which were not completed, but it is important to evaluate and/or justify why this is the case.

3. Outline any open problems or future plans. Rather than treat this only as an exercise in what you *could* have done given more time, try to focus on any unexplored options or interesting outcomes (e.g., "my experiment for X gave counter-intuitive results, this could be because Y and would form an interesting area for further study" or "users found feature Z of my software difficult to use, which is obvious in hindsight but not during at design stage; to resolve this, I could clearly apply the technique of Smith [7]").

Bibliography

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

[2] ——, *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/.

[3] 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.

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

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

[6] 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.

[7] 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/.

[8] 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.

[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/.