

Candidate Number: 164776

OpenFlow and Software-Defined Networking

BSc (Hons) Computer Science  
Supervised by Dr George Parisis  
Informatics Department  
2019

*The Design and Development of an OpenFlow Toolkit for deploying and experimenting on Software-Defined Networks.*

*This report is submitted as part requirement for the degree of BSc (Hons) Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.*

# Summary

This report details the design of a toolkit that allows the user to set up an *OpenFlow* network. It introduces the concepts of *OpenFlow and Software-Defined Networking* and introduces the toolkit, which is primarily designed for companies to experiment with simulated networks that emulate real data centre topologies in a *Virtual Machine,* that contains and utilises powerful networking experimentational software.

As well as detailing the requirements, specification and development process of the toolkit, this report will cover the findings of research performed in the *Software-Defined Networking* field and discuss the key concepts as well as several key *OpenFlow* projects that have either inspired or featured in the toolkit.

Through researching the field, it became clear that there is an abundance of software, open-source and tailor-made for businesses alike, and so the vision for the *OpenFlow* toolkit became abstract; rather than trying to tailor the toolkit to support specific software, allowing the user to import the *OpenFlow* software of their choice means the toolkit can theoretically support any *OpenFlow* software that executes on the command-line.

The toolkit will allow its users to create a *network topology*, import it and custom *OpenFlow* software of their choosing into the toolkit, transfer it to the VM via *SFTP* and remotely access the VM via *SSH Terminals* where the user can input commands. This can allow a user to easily conduct their own experiments, streamlining and simplifying the network deployment process.

While the toolkit is based on abstraction from specific software, the toolkit directly supports deploying a simulated network with *Mininet*, which can create expansive virtual networks with real code on a single machine.

The toolkit also directly supports *Veriflow*, which acts as a verification layer between a remote controller and the rest of the network. *Veriflow* can be used to test for bugs in a network, such as black holes or ARP spoofing.

This report also discusses how the toolkit was tested and walks through several networking experiments that are performed with and through the toolkit.

By learning and discovering about the *SDN* field, a conclusion is that the field is vast, yet can be very specialised. There are many great general *OpenFlow* projects and controllers, yet many businesses tailor-make plans that suit them. For smaller businesses and academia that do not have that luxury, being able to easily and quickly research and test software in the field to discover what best suits their needs is predominant, and so the *OpenFlow* *Toolkit* is designed to simplify the deployment and experimentation of *OpenFlow* networks.

Table of Contents

[Summary 2](#_Toc7404501)

[1. Introduction 3](#_Toc7404502)

[2. Professional and Ethical Considerations 4](#_Toc7404503)

[3. Related Work 5](#_Toc7404504)

[4. Requirements analysis 8](#_Toc7404505)

[4.1. Target Users 8](#_Toc7404506)

[4.2. Scope 8](#_Toc7404507)

[4.3. Non-Functional requirements 9](#_Toc7404508)

[4.4. Functional requirements 10](#_Toc7404509)

[5. Project specification 10](#_Toc7404510)

[5.1. Project details 11](#_Toc7404511)

[5.2. Assumptions and Dependencies 12](#_Toc7404512)

[6. System design and development 13](#_Toc7404513)

[6.1. System design 13](#_Toc7404514)

[6.2. System development 15](#_Toc7404515)

[6.3. System testing 17](#_Toc7404516)

[6.4. Discussion of principles 17](#_Toc7404517)

[6.5. System evaluation 18](#_Toc7404518)

[7. Conclusion 22](#_Toc7404519)

[8. References 24](#_Toc7404520)

[9. Appendices 26](#_Toc7404521)

[9.1. Appendix A – Project Log 26](#_Toc7404522)

[9.2. Appendix B – Setup Guide 27](#_Toc7404523)

# Introduction

The primary aims of this project are to explore and learn about Software-defined networking (SDN), and then create an OpenFlow toolkit that can implement OpenFlow network deployments in a Virtual Machine (VM) that mimic actual data centre topologies. These virtual networks will be supported by routing algorithm(s) and controlled via an OpenFlow controller. The project involves experimentation in these deployments, to gain an understanding of the controllers and algorithms used as well as understand all the parts of an OpenFlow network topology (e.g. how a controller routes packets in a given network and pushes entries into a flow table to immediately update the network’s behaviour).

Since OpenFlow’s creation, it has been massively expanded and is used by many large companies (e.g. Microsoft [1] and Google [2]) as they program their own networks. It remains a hot topic in computer networking and experimenting with it via the creation of a virtual data centre network will help increase knowledge of how businesses deploy their own networks in real-life situations.

As such, this project is primarily focused on learning about the many facets of OpenFlow and SDN and then creating a unique network, deploying it in a virtual machine with its own routing algorithm and OpenFlow controller, and be able to push flow entries onto it and update the network at will. Another aspect to consider is experimenting with software verification tools (e.g. *Veriflow* [3]) to statically and even dynamically verify OpenFlow network deployments.

This report will fully cover the planning, design, development and experimentation of the OpenFlow toolkit. The *Related Work* chapter will cover several key concepts and applications that are relevant to this field and are connected to the toolkit in some way. The *Requirements analysis* chapter establishes the target users that the toolkit will be developed towards and would be most likely to benefit from using it to set up their own networks, as well as setting the toolkit’s scope, and establishing the key functional and non-functional requirements to consider while designing the toolkit.

The *Project specification* chapter will give an overview of the toolkit’s interface, describe the functions of each toolkit component and establish any assumptions and dependencies that could constrain the toolkit.

The *System* chapter will cover the design and development of the toolkit, describe how the toolkit has been tested for bugs during its development, discuss some of the key design principles and choices that have been made, and walk through experiments that showcase the unique features of the toolkit.

Lastly, the *Conclusion* will summarise what has been achieved and learned throughout the project and give a critical evaluation of the toolkit, including what was missing or could have been developed better, and how the toolkit would be received by its target group. It will be followed by the *References* and the *Appendices,* which contains the project log and a setup guide for the toolkit.

# Professional and Ethical Considerations

The project student will have due regard for public health, privacy, security and wellbeing of others and the environment, have due regard for rights of Third Parties, conduct all professional activities without discrimination, and will promote equal access to the benefits of IT.

This project is based on learning and experimenting with mostly software that is freely available online to download and use for educational purposes. One consideration is not to break point 2(d) in the *BCS Code of Conduct* [4], which is to ‘ensure that you have the knowledge and understanding of Legislation and that you comply with such Legislation, in carrying out your professional responsibilities.’ To ensure that the project complies with any necessary Legislation, any software that may be used should be first checked to ensure that the software can be used for the purposes of this project.

This project should also ensure to keep awareness of technological procedures and standards, as mentioned in point 2(c). The project is focused on looking at cutting-edge networking software, so this will be easy to maintain.

Point 2(e) will be fulfilled through communication with the academic and technical supervisors, e.g. asking for opinions on certain aspects of the project topic, getting clarification on topics, and through feedback on project drafts and the draft report.

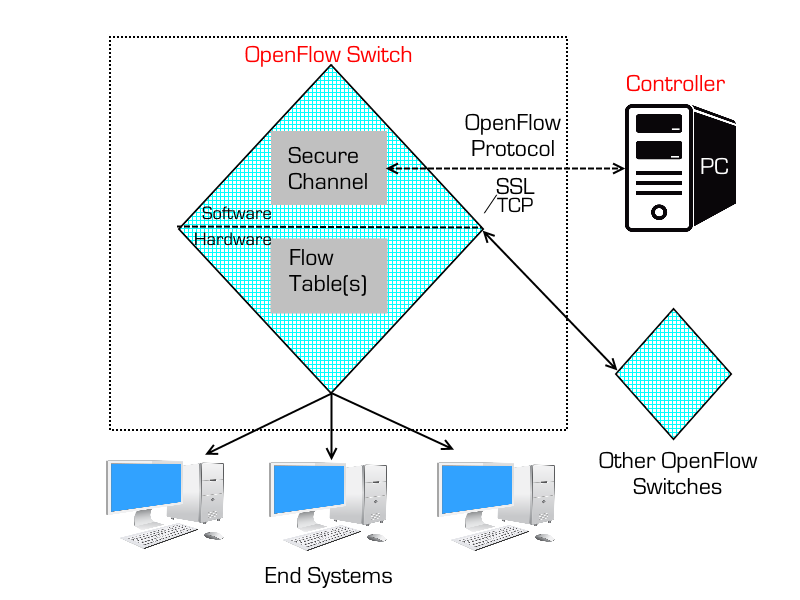
The project student will only undertake work that is within their professional competence, and will avoid injuring others or their property, reputation, or employment by any means. The student will reject and will not make any offer of bribery or unethical inducement.

The project student will also adhere to all the points in sections 3 and 4 of the *BCS Code of Conduct*, fulfilling their duty to relevant authority and to the profession.

# Related Work

Software-Defined Networking (SDN) is defined as ‘The physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices.’ [5] This allows the network control plane to be customised and the core infrastructure to be abstracted, which is the basis behind the OpenFlow protocol. SDN is the dynamic architecture that allowed OpenFlow interfaces and software to take off in the networking field.

OpenFlow was initially designed as a way ‘for researchers to run experimental protocols in the networks they use every day’. [6] The primary function of OpenFlow was to serve as a programmable network, based on an Ethernet switch with an internal *flow table(s)*. Flow tables are stored within an *OpenFlow Switch*, which has links to hosts, other switches and a *controller*. An OpenFlow Switch [7] is comprised of three parts (see Figure 1); the internal *flow table*, a *secure channel* that the switch uses to connect to the controller, and the *OpenFlow protocol* that provides a way for a controller to securely communicate to the switch and allows for packets to be transmitted between the switch and the controller. While OpenFlow has been expanded upon since its creation, the switch still has the same core parts.

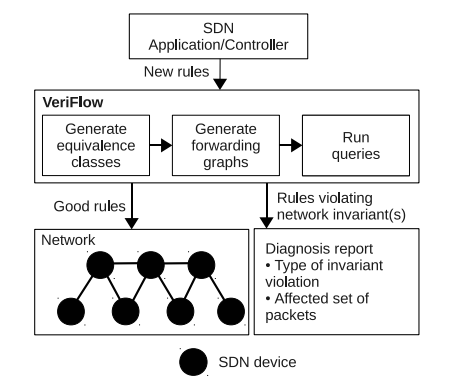
A flow table is in control of directing packets to and from connected hosts, and it does this based on the *flow entries* that the table holds. A flow entry contains a variety of different settings and actions that affect how the switch processes the flow. The controller can use the OpenFlow protocol to insert, modify and remove flow entries, either at will or automatically in response to incoming packets.

*Figure 1: An OpenFlow Switch and its connections. It transmits packets to end systems, or to other OpenFlow Switches in the network depending on the rules in its flow tables.*

The remote controller is used to manage the network. Just as there can be multiple switches in a network, there can also be multiple controllers managing the same switch, for extra performance and/or efficiency. There are a variety of controllers to choose from in different languages such as *Python*, *Java* and *C++*. Learning and testing for this project has mostly been performed using *POX* [8] as the remote controller of choice due to its stability, widely enjoyed support and range of network components to experiment with, although the toolkit will be designed to support other remote controllers.

An OpenFlow network is ran on *Mininet*, which itself is ran on a VM. (See *Assumptions and Dependencies* for details.) *Mininet* [9] is a network emulator that supports topologies that contain hosts, switches, links and controllers, and it specifically supports OpenFlow switches. It runs on Linux network software and its website has a Linux OS with *Mininet* pre-installed, which is the VM that has been used to test this project. Importantly, it supports custom topologies and connects hosts and switches through virtual ethernet pairs, meaning it can efficiently support large-scale simulated networks (the website states it can support at least 4096 (212) hosts and switches in a single OS kernel).

*Veriflow* is a very important concept in SDN, as software-defined networks can become exceedingly complex and large. Because of this, as a network is modified, it is necessary to ensure that the network is free of bugs, but as the network can automatically update itself this becomes very difficult. *Veriflow* acts as a layer between the controller and other network devices (see *Figure 2*), checking for network-wide invariant violations as flow entries are added, updated and removed. *Veriflow* has its own proxy port which the network communicates with when it tries to send data to the controller. *Veriflow* analyses the data packets, sends them to the controller and when the controller sends back packets to their intended destination and tries to set or update flow entries in the network, *Veriflow* analyses these entries to ensure they would not cause a network violation. This gives the user a critical, extra level of control over their network, which is why *Veriflow* was specifically chosen to be implemented into the toolkit.



*Figure 3: Veriflow acts as a verification layer between the Controller and the rest of the network, checking data coming in and out of the controller for potential network violations.* [10]

One of the advantages of *Veriflow* is that it can be added to a network independent of the other parts of the network. (This also retains the simplicity of the toolkit, as *Veriflow* is given its own section that the user can ignore if they so choose.) All it needs to know is the network’s topology, which the toolkit parses from the user’s graphically-created topology. *Veriflow* will act like a passive controller, letting new rules and packets be transmitted between a controller and the local section of the network (an *equivalence class* (EC)), only blocking transmissions that would cause a bug. It does this without sacrificing much performance by splitting the network into *equivalence classes*, which are local slices of the network that a given set of packets would affect (usually only a small section of a large network), meaning there is less to analyse with each change. *Veriflow* then builds individual *forwarding graphs* for every altered EC, which represent the network’s forwarding behaviour, and then traverses these graphs to determine the status of an invariant(s).

# Requirements analysis

The purpose of this project is to gain understanding of the underlying concepts of SDN and OpenFlow as well as explore the possibilities of the field through experimentation with popular OpenFlow software, which would provide great knowledge and hands-on experience in the networking field. To efficiently experiment with various OpenFlow software, the primary focus of the project is to create an *OpenFlow Toolkit* that can be used to set up an OpenFlow network inside of a Virtual Machine with *Mininet* [10] and the OpenFlow controller of the user’s choice.

Most target users are concerned with a few key points. These general key points are covered in the non-functional requirements. The toolkit itself is a means to an end; an end user is more interested in setting up their network and performing experiments than learning how to use the toolkit and trying to work around it. The toolkit has been developed with consideration to this, which is why key focus has been on abstraction and a simple interface. Some users will also have specific needs which are covered in the functional requirements.

This chapter will first establish the target groups that would be likely to use the toolkit and its production is aimed towards, followed by establishing the scope of the toolkit’s development. Then, there will be sections dedicated to predicting the target users’ needs, as well as analysing functional and non-functional requirements that the toolkit should achieve.

## Target Users

The target groups of this project are companies, universities and organisational bodies who have expansive enterprise networks, are interested in the extra control and performance benefits that come with SDN [11] and want to implement or experiment with OpenFlow in their network. The organisation could be interested in testing some of the many OpenFlow Projects that are freely available online [12] or tailor-making their own software, but are unsure of what would be the fastest, simplest or most stable choice for their network and are interested in efficiently testing and experimenting with different projects in a simulated network.

## Scope

The toolkit is intended to simplify and streamline the setup of an OpenFlow network. It contains a variety of options to assist with the deployment process. Currently, to set up an experiment, one must open an emulator, start up a virtual OS with *Mininet* installed, log in, start the GUI, open several terminals to start *Mininet* and a remote controller of choice and optionally open an SSH client to access the virtual network remotely.

The toolkit will enable users to set up a custom OpenFlow network. The user will step through the toolkit’s interface to set up each component of the network within the toolkit, such as setting the VM details to connect to, providing a remote controller of the user’s choice, creating a network topology and then deploying the network, all within the toolkit.

The scope of the project is focused on maximising support of external OpenFlow software. If the toolkit restricts the user from testing certain OpenFlow software or their components due to its design or lack of support, the user would lose any benefit the toolkit would be giving and abandon it.

The planned approach to ensure support for many, different pieces of software is to abstract away from supporting specific software. Rather than building the toolkit around many programs, the toolkit has a clean and simple interface that allows the user to import the SDN Controller of their choice and create their own network topology. This approach means that the toolkit itself acts as an abstraction layer for the software’s that used to set up the OF network (e.g. Mininet, the remote controller, the Secure Shell (SSH) terminal that connects to the VM, etc.) and allows the user to configure each part of the network within the toolkit.

## Non-Functional requirements

*Efficiency* – The primary advantage of using the toolkit instead of working with each individual part is efficiency. The toolkit automates several parts of the setup process, namely transferring network components from the user’s PC to the VM and then executing those components. It acts as a middle man between each network component and *Mininet* by opening SSH terminals that execute each component with the user’s chosen arguments.

The topology editor will also speed up the creation of a custom network topology as the GUI will provide users with an easy way to create their topologies, which are saved and automatically parsed into files that *Mininet* or *Veriflow* can interpret. This is considerably more efficient than writing a topology file manually.

*Simplicity* – If the toolkit is too complicated to use, it would be abandoned. Companies that are considering the SDN field are spoilt for choice on what software to test for their desired needs. The toolkit is simple, as it lets the user import their own chosen software and execute it with their own command-line arguments. The user can import an existing topology, or create their own, and execute *Mininet* with their chosen arguments. The mantra when creating the toolkit was that by not supporting any specific OpenFlow software (aside from *Mininet*, and *Veriflow* which has a specific use), the toolkit supports every piece of OpenFlow software. The toolkit never directly interacts with any other software, it simply transfers them to the VM and invokes them in a remote terminal with the user’s given arguments. This makes the toolkit easy to learn and simple to use. The topology editor is also a drag-and-drop GUI, which feels intuitive to create topologies with.

*Customisability* – As mentioned in the *Simplicity* section, by not supporting specific OpenFlow software, the toolkit indirectly supports most or practically all OF software in practice. This gives users the freedom to customise their own OpenFlow networks with the software of their choice. Users can also pick and choose the sources of each component; for example, a user can create a topology, save it on their PC, import it and use it in *Mininet* and then run a remote controller that already exists on their VM.

*Wide range of support* – This has been indirectly covered in the last two points. By abstracting from providing support to specific software, the toolkit in theory supports every controller that can be invoked through a terminal.

## Functional requirements

*Remotely access a VM* – Typically, OpenFlow networks are deployed on a *Linux VM* which has *Mininet* running (see *Assumptions & Dependencies*). The user should be able to navigate the toolkit’s graphical interfaces to customise their network setup, and then remotely connect to their VM via an SSH terminal. A section of the toolkit will allow the user to set the VM’s settings, including username, password, and the IP address and port of the VM’s network adapter.

*Custom remote controller* – The toolkit should be able to support custom OpenFlow software, especially custom OpenFlow controllers. For users to experiment with their custom networks, they need to be able to import their desired software that they wish to perform experiments with.

*Choose network component locations* – A user may wish to import software they have been working on at their PC. Another user may already have their desired software residing in the VM and wants to invoke it directly. And another user may want to use a remote controller that exists on their VM but wants to import a topology file they just created in the toolkit. The toolkit should have a simple toggle that states where the network files that the user wants to invoke exist, whether on their PC or on their VM, and the user must be able to toggle it between each step of the setup process if desired.

*Create network topologies* – Users should be able to graphically create their own network topologies within a dedicated interface. The topology editor should allow users to drag and drop hosts and switches as well as create links between them. The topology editor should also be accessible for networking beginners; only people who are experienced with *Python* [13] and *Mininet/Veriflow* could program their own topologies, so a networking administrator who wants to perform experiments on a variety of network setups quickly should be able to make a graphical topology in the topology editor, that is then parsed into a *Mininet and/or Veriflow* topology file.

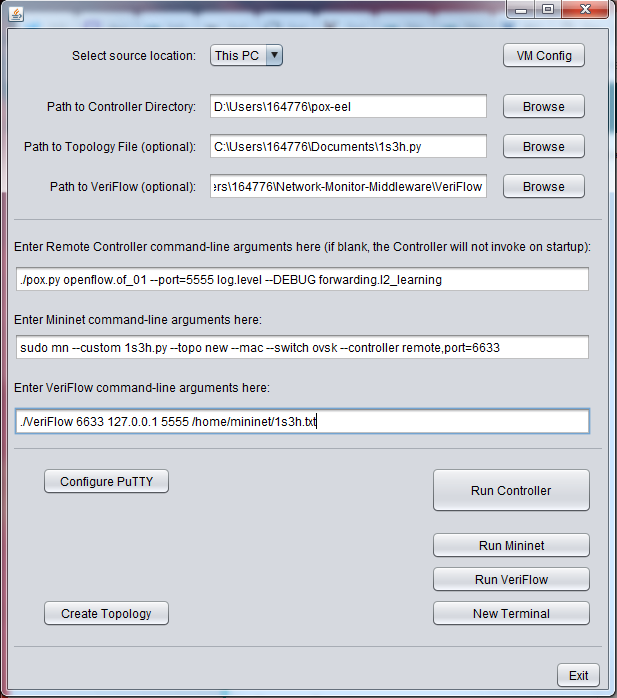
*Test buggy networks* – Users may wish to create an intentionally buggy network for experimental reasons, e.g. they may have developed their own controller which has glitches and are trying to debug it. The toolkit will implement *Veriflow* [3] for this purpose, which is discussed in-depth in the *Related Work* section.

# Project specification

This section will fully cover the project details and any design or implementation assumptions or dependencies.

## Project details

The toolkit (see *Figure 3*) allows the user to remotely connect to a VM. It has a configuration section where the user can input the VM’s details, namely the IP address and port to connect to the VM, and the username and password to remotely log into the VM.

**

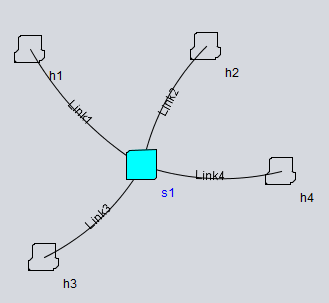
*Figure 3: The OpenFlow Toolkit.*

The toolkit supports *Mininet* directly as it is an industry-standard, open source networking software that gives the user wide support for OpenFlow networks and is relatively easy to install and learn. The user will create their network in *Mininet*.

The user can decide to set up the network with components (the remote controller, the network topology file, and *Veriflow*) that are on the user’s PC, or already exist on the VM. If the components are stored on the user’s PC, the toolkit will transfer the components through SSH when the user runs each component. If they’re stored on the VM, the user can specify in the toolkit where the components exist on the VM and the toolkit will call to their location in the SSH Terminal.

Text fields for each component will exist that allow the user to specify command-line arguments for each terminal that runs each component. When the user runs a component, the terminal will automatically execute the command that’s stored in the text field, which typically starts a component with the user’s custom arguments. If the user does not specify any arguments and leaves the field blank, the terminal will not execute the respective component and will stay pending for further input.

The interactive network topology editor is a graph-based GUI that allows the user to drag and drop hosts, switches and links (see *Figure 4*). Users can toggle between placing a new host or switch in the menu. Users can also right-click on a host, switch or link to open a pop-up menu that contains relevant settings (e.g. host and switch addresses). The controller has its own dedicated menu that lets the user configure its settings, namely which switches it will be linked to. The user will be able to save their topology in formats that are supported by *Mininet* and *Veriflow* respectively.



*Figure 4: An example of a star topology* [14] *created in the topology editor with hosts, links and a central switch.*

## Assumptions and Dependencies

The toolkit relies on some other applications existing for it to work as intended. These application dependencies are detailed below.

*Mininet* – The toolkit is built around *Mininet*, which means it is assumed that the user should have *Mininet* installed on their VM. If the user is deploying their network on software other than *Mininet*, the toolkit would lose its topology functionality.

*Virtual Machine* – It is assumed that the toolkit is remotely connecting to a VM that has an open port that the toolkit can connect through via SSH. If the user has an unconventional setup, the toolkit may not be able to remotely connect with a terminal(s).

*PuTTY* – In order to transfer files from the user’s PC to the Linux VM, the toolkit must support *PuTTY* [15], specifically the *SFTP* (SSH File Transfer Protocol) executable file that comes with *PuTTY*. This means the user must have *PuTTY* installed on their PC before transferring files.

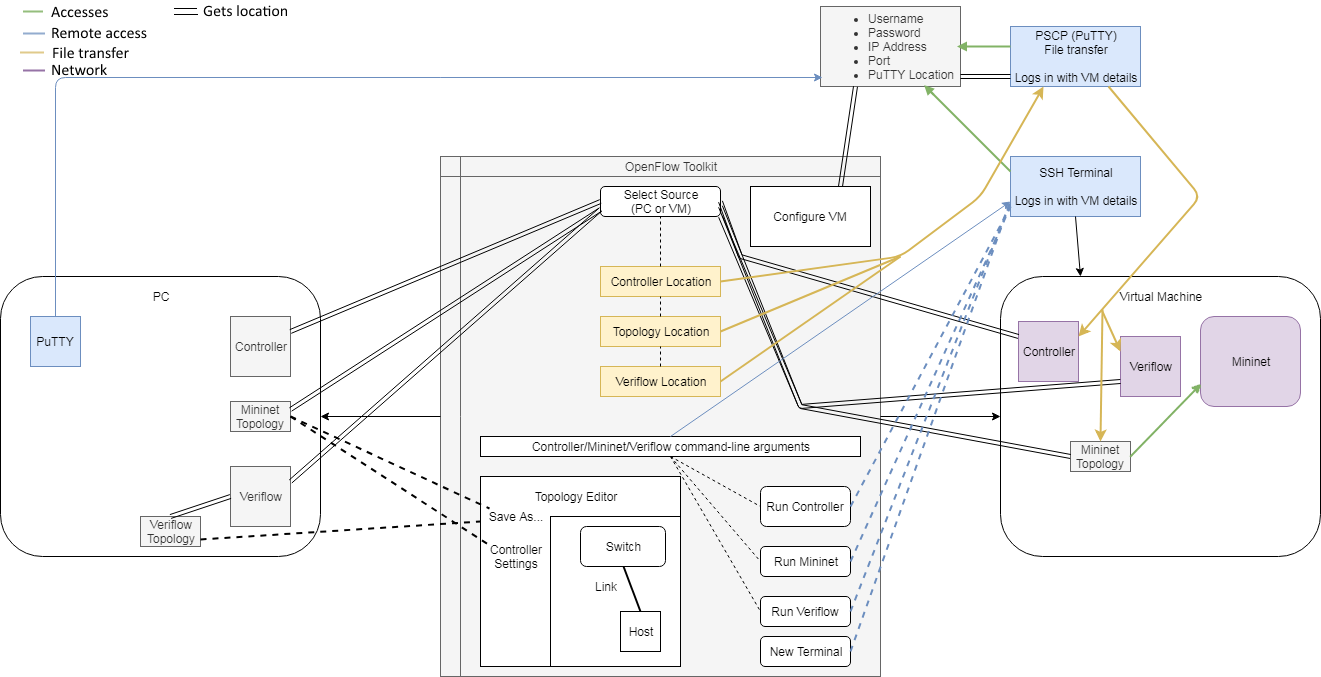
# System design and development

This chapter will cover the design and development of the toolkit. The design section will discuss which programs were used to create the toolkit and why they were chosen, and the development section will review the process of building the system.

## System design

The toolkit’s creation was chosen to be done in *NetBeans* [16]. This is for several reasons; firstly, *Java* [17] is the most commonly used language in this degree which makes developing under time constraints less stressful. Secondly, *Java* enjoys a considerable number of external libraries that add functionality, of which some key libraries will be discussed later. Thirdly, *NetBeans* contains a design section which is incorporated with *Swing* [18] that simplifies designing the GUIs for the toolkit. As this project is being developed under a time constraint, creating the GUIs visually was an essential decision so as not to risk sacrificing the quality of the rest of the project.

Below is a component diagram that shows the toolkit’s components and how they link to components in the user’s PC and VM:



*Figure 5: A component diagram of the toolkit’s working parts. A colour key is in the top left.*

To create the interactive network topology graph editor, several libraries were considered. Originally, the toolkit was attempted to be designed with *JGraphX* [19] due to its compatibility with *Swing*. However, *JGraphX* does not have much interactive functionality, and is primarily used to make static graph. Next, the library *GraphStream* [20] was considered as it is a dynamic graph library. However, development with it was slow because it lacked in-depth documentation and support, meaning development with *GraphStream* had to be abandoned as well.

The third library that was chosen to develop the interactive graph editor was *JUNG* [21] (Java Universal Network/Graph Framework). This library has extensive documentation and support and contains frameworks that can be extended and overridden for easy customisation (see *Discussion of Principles*). Development with *JUNG* is discussed in detail in the next section.

To allow the toolkit, written in *Java*, to connect to a VM, the toolkit needed to contain a library that provides SSH functionality. On the VM side, the VM that is located on the *Mininet* website [10] uses *SSHD* (OpenSSH Daemon) [22] to process incoming SSH connections. Several libraries were tested with the toolkit, but the library that was implemented into the toolkit is *JSch* (Java Secure Channel) [19] as it is the most robust *Java* library with SSH functionality.

As mentioned in *Section 4.3*, the toolkit supports *PuTTY* [15] for SFTP functionality. This means that any files and folders that are to be transferred from a PC to a VM, such as a remote controller or a user’s topology file, are transferred to the VM via SFTP *before* the toolkit directly connects via *JSch*. This is because the SSH terminal logs in to the VM and at that point, cannot ‘see’ the user’s home PC.

To add network verification to the toolkit, direct support for *Veriflow* [3] was implemented into the toolkit. Like with the remote controller support, the toolkit does not directly interact with *Veriflow* itself, but does allow the user to save network topologies they create as a *Veriflow* topology, which the user will be prompted to select and transfer from their PC before running *Veriflow* through the toolkit.

However, it was not possible to get an industry copy of *Veriflow* for this project. Their website [24] is tailored towards companies and does not have an openly available version of their software. Instead, the project will be using a research copy of *Veriflow* [25] which is openly available for research purposes but may contain some bugs. The process of implementing *Veriflow* is detailed in the next section.

The toolkit will need several classes for each GUI. One will be the main GUI and will have a main method that the user can run to start the toolkit. This GUI will be the primary interface and will provide links to the rest of the toolkit.

The topology graph editor will have its own window, and the user will be able to use their cursor to create topologies containing hosts, switches and links, as well as connect the switches to a controller. The user will be able to save their topology through a menu that will open a dialog frame.

Each component of the topology will require their own popup menus to alter their settings, such as their name, and e.g. for a host, its IP and MAC addresses. *Java* libraries will also need to be imported to add functionality for IP and MAC addresses so that the user does not accidentally input the wrong characters into those text fields.

To import the controller, the user will be able to select a folder through a dialog window and its location will be stored in a text field adjacent to the browse button. This will also be repeated for *Veriflow* as well as for the topology file (although the user will select a single *Python* or *Mininet* file instead of a folder). If the user is going to invoke their next component from the VM, the browse buttons will be disabled and likewise, if a respective component location text field is empty, the run button for that component will be disabled.

There will be a Run button for each component (namely the controller, *Mininet* and *Veriflow*). When this is executed, the toolkit will first check whether the component is located on the user’s PC or on the VM. If the former, the component will be transferred via SFTP, and then the toolkit will SSH into the VM and then check if the user has specified to run the component with command-line arguments. If they have, the toolkit will execute the component on the VM with those arguments, if not, the toolkit will leave the terminal blank, ready for the user’s input.

## System development

The main GUI was developed in *Swing*, which was relatively simple to design. There were several challenging parts to developing the toolkit, namely developing the topology graph editor, parsing the topology into a *Mininet* or *Veriflow* topology file respectively, transferring files from the user’s PC to a VM, connecting to the VM through a visible SSH terminal and getting the research copy of *Veriflow* to work with the toolkit. This section will detail how each of those parts were developed and will highlight important design choices that were made, while the *System Testing* section will explain how each part of the toolkit was tested for bugs.

*Developing the topology* – As mentioned in *Section 5.1*, the toolkit had to have its graph visualisation library changed twice before finding a library that fit the specific needs of the toolkit. *JUNG* was chosen as the visualisation library as it contained extensive support for interactive graphs and graph editing. When developing the topology, a graph editor example project that is freely available on the JUNG website [21] was used and modified for the toolkit, as it has an effective implementation of a basic JUNG graph editor, which the toolkit expanded on and was converted into a full network topology graph editor. Classes which were based on the example project are appropriately referenced in the code.

From this, it was possible to build a topology graph editor. The vertex was split into two for hosts and switches, with a different image and different popup menus for each. To accurately portray IP and MAC addresses in the host/switch popup menus, the ‘*java.net.Inet4Address’* [26] class was imported for IP addresses, and a *‘RegexFormatter’* class that imports classes from the *‘java.util.regex’* package [27] will support hex values for MAC addresses. Several menus were added, such as a ‘Save As’ menu which has options for *Mininet* and *Veriflow*, as well as a ‘Controller Setup’ menu. It was decided to keep the controller outside of the topology, as a large majority of topologies do not need more than one controller and so it should have its own independent section. The controller menu has options for its name, port that *Mininet* will use to connect to the controller, the controller type (remote for the user’s own controller or OVS if the user wishes to use the OVS controller [28]), the controller protocol (TCP by default, or SSL), the controller’s IP address (typically *localhost*) and a list of switches that are already in the graph that the controller can be linked to.

*Parsing the topology* – Once the user creates their topology, they need to save it. To convert a graphical topology into a *Mininet* or *Veriflow* file, the topology class contains methods that scan the vertices and edges in the graph as well as their settings and parses them into Python language to create a *Mininet* topology, or to the proprietary, bare-bones format that *Veriflow* uses for its topology. This is performed by concatenating the details of each vertex and edge, as well as the controller settings in the case of the *Mininet* topology, into a single script string. This method of parsing is used as the task of converting *JUNG* collections into specific languages is a unique problem, which meant parsing it through a concatenated string was simpler to code than trying to find a library that could do it instead. Each setting in the graph has been tested extensively against the parser to ensure it will always produce accurate topology files, as detailed in *Section 5.3.*

*Transferring files to the VM* – The toolkit uses *PuTTY’s SSH File Transfer Protocol (SFTP)* application to transfer files from the user’s PC to a VM. This is executed by getting PuTTY’s location on the user’s PC, starting command prompt and pointing it to SFTP in the PuTTY folder. Files are only transferred when the user has their source location set to their PC, and after the user has specified where their controller or *Veriflow* folder, or topology file is located. However, when the toolkit is transferring a folder, the DOS line endings (carriage return plus line feed) are not properly converted to Unix line endings (line feed) which causes the files in the folder to not work once opened. To fix this issue, after a folder is transferred to the VM, *dos2unix* [29] is installed (if not already installed) on the VM and is ran on the folder to convert the files’ line endings.

*Connecting to the VM* – There were some issues getting *JSch* to connect to the VM, namely getting a visible terminal to appear was not possible. Like with the topology graph editor, other libraries were tested, including trying to SSH into the VM with *PuTTY*, but ultimately *JSSHTerminal* [30] was selected. This library itself uses *JSch* to connect via SSH to the VM and opens an *xterm* emulator that acts as the terminal which the user can input commands into.

Before the user can input commands, the toolkit must check if it needs to transfer any files to the VM. As mentioned before, this process is done with *PuTTY*’s SFTP functionality.

*Developing Veriflow* – Working with a research copy of *Veriflow* caused some problems that were complex to resolve. The main issues with implementing *Veriflow* were understanding how its topology files worked, as there was no documentation on how they are parsed, and on how to interpret the ARP table. Before being executed, *Veriflow* must be created with an ARP table of the topology included in its source folder. When the user’s topology is parsed to a *Veriflow* topology, the toolkit will ask the user to save an ‘arp.txt’ file into the *Veriflow* directory, as this is necessary to build the *Veriflow* executable file and run the program. Additionally, a change had to be made in the source code itself; even when running *Veriflow* with the correct topology and ARP table, it claimed that the ARP table was poisoned when the topology attempted to send data packets to and from the controller, as these packets did not have IP/MAC source or destination addresses set. To resolve this, a method that checks incoming packets had to be modified so that even if the destination IP/MAC addresses don’t match the ARP table, if they are 0 and therefore empty, it is not necessarily an ARP poisoning fault and the packets can continue.

## System testing

To ensure each section of the toolkit works as intended, extensive testing was performed at each step of the development process. This type of software is mostly tested through trial-and-error as each component is developed; for example, every time the topology editor is updated, create and save a topology to check that it outputs correctly. Every time the import or remote access features are updated, connect to a VM and transfer files, etc. Therefore, this section is going to cover general areas of the toolkit where extensive, repetitive testing has been performed throughout the toolkit’s development. Other small-scale tests were performed after each piece of the toolkit was programmed, again via a trial-and-error approach.

The key areas to test are the topology editor, parsing a topology, connecting to the network, transferring network components, and executing a test network with a controller and *Veriflow*.

*Topology editor* – Testing the topology editor is straight-forward, as it is a drag-and-drop interactive GUI. Being able to create, move, rename and delete hosts, switches and links was tested, as well as checks on their fields to ensure key fields save as intended and only accept the correct character (i.e. hex values for MAC addresses, etc).

*Parsing a topology* – Each feature of the topology graph editor must also be tested in the *Mininet* topology file so that each is correctly implemented and interpreted by *Mininet*. *Veriflow* topologies do not need as much testing, as they only implement host and switch IP addresses as well as their routes, and do not store other details.

*Connecting to the network* – This involves ensuring that the VM has an open network adapter and that the correct details are stored in the VM config window in the toolkit. Several other features were tested, such as transferring individual files as well as folders via *PuTTY*, ensuring the automatic deployment of network components is performed correctly whether the component originates from the user’s PC or the VM and that the *dos2unix* conversion process does not cause unintended changes.

*Executing a test network with a controller and Veriflow* – Specific experiments are discussed in *Section 5.5*, but in terms of testing the toolkit works as intended, each time a major change was implemented, the test was to create a new topology and save it, transfer it along with a controller and *Veriflow*, then set up a test network and run checks that it works as intended. This process was performed constantly through the development of the toolkit to ensure that no unintended side-effects occurred during development.

## Discussion of principles

This section discusses some of the technical choices made in the development of the toolkit. It will cover some of the more interesting and important choices and justify them, such as how some data structures behave, how different classes interact, etc.

*Singleton classes* – The toolkit uses singleton classes, which only allows a single object of a class to persist. Singleton classes are used in the toolkit for the classes that open GUIs (except for the main GUI). When a GUI is closed, it needs to continue to exist so that the toolkit can access its field values in other classes. In the code, when a GUI is closed (aside from the main GUI, which exits the program), it is set to be invisible but is not disposed of. Each GUI class has a method that either gets its instance or initialises a new instance if the method is being called for the first time. This allows other classes to access values that a user sets in one GUI even after it is ‘closed’ by the user.

*JUNG Transformers & Renderers* – One of the advantages of *JUNG* is that it comes with a range of pre-made Transformer and Renderer classes (e.g. *VertexIconShapeTransformer, BasicVertexRenderer*, etc.) that can be invoked and overridden. *JUNG’s* classes have several pre-sets to fit specific purposes, which makes *JUNG* support a considerable number of graph variations and needs as standard. When a user wants something specific, they can override specific methods in any of the renderers or transformers and make custom methods. For example, the toolkit overrides the *setVertexFillPaintTransformer* with a custom Transformer that paints vertices with different colours depending on whether they’re a host or a switch.

*Vertex & Edge classes and factories* – Vertices and edges are created in factory classes that add the respective vertex or edge to the graph. They are populated with default values and added to *Collections* in the graph that are called when parsing the graph into a topology file.

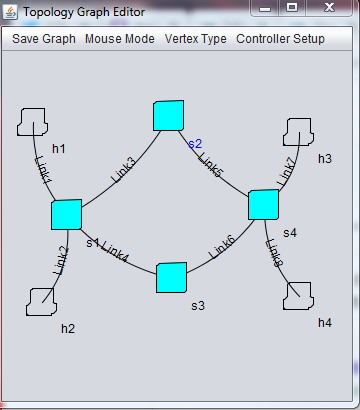
Additionally, both hosts and switches use the same *Vertex* class. The class has a field *type* which will be initialised as either ‘Host’ or ‘Switch’. When another method in the program accesses a *Vertex* instance, it will use the *type* getter to decide how to process the instance.

## System evaluation

Two experiments will be performed through the toolkit to give examples of what a user can perform by setting up and deploying their networks through it. Each experiment will demonstrate some of the key capabilities of the toolkit; the weighted-graph loop topology experiment demonstrates some of the features of the topology graph editor, notably the ability to set the bandwidth of a link. The *Veriflow* ARP spoofing experiment demonstrates how *Veriflow* can be used to verify the integrity of an OpenFlow network and protect it from attack.

Before moving to the experiments, it is important to preface why these specific experiments were chosen to be discussed in this report. A system like this toolkit is tricky to effectively evaluate, as its primary purpose is to transfer and deploy other software. These experiments utilise the key features of the toolkit, which are its topology editor and *Veriflow*, the latter of which has been patched to fix bugs and packaged with the toolkit. Experiments that do not feature either of these could be discussed here, but in general the toolkit would have virtually no bearing on them other than being a convenient way to set the experiments up, with possible exceptions. Nonetheless, these experiments showcase what is possible *through* and specifically *with* the toolkit due to the use of its exclusive features, while other, more general experiments would be specified as being performed only *through* the toolkit.

*Experiment with a weighted-graph loop topology* – This experiment will use the topology editor to create a *Mininet* topology with four switches and four hosts. The switches will be linked in a loop, and each switch will have a link to one of the hosts. One of the switches will have its links be weighted, by setting their bandwidth in the topology editor to 10 *Mbit/s*. The other links will have unlimited bandwidth. See *Figure 6*:



*Figure 6: The loop topology that has been created. The links emanating from switch 2 have had their bandwidth restricted to 10 Mbit/s.*

Once the topology has been created and saved, it is selected for import in the toolkit and is mentioned in the command-line arguments that will be invoked when *Mininet* is ran.

Networks with loops can cause issues, as a data packet running along the loop can do so repeatedly and will not branch off the loop to find its destination.

Dedicated code is required to detect network loops. To process this loop topology, the experiment will use *POX*’s *‘openflow.spanning\_tree*’ component [31] which in turn uses the ‘*openflow.discovery’* component to find a view of the network topology, create a spanning tree from it and then disable ports that are not on the tree. It will also use the *‘forwarding.l2\_multi’* component, which benefits from OpenFlow discovery to learn the entire topology (i.e. when one switch learns the location of a host, the entire network learns) by setting up forwarding tables.

This experiment will attempt to discover two outcomes; first, if *POX* is able to successfully detect the loop and disable or bypass the loop. Second, if *POX* can detect that some of the links have a limited bandwidth and use the unrestricted links.

The command used to execute *POX* with the desired components is:

/pox$ ./pox.py log.level --DEBUG openflow.spanning\_tree --no-flood --hold-down openflow.discovery forwarding.l2\_multi

*‘No-flood’* pre-emptively disables flooding on all ports when a switch is connected and will re-enable ports that do not cause a loop after. *‘Hold-down*’ stops flood control from being altered until *‘openflow.discovery’* has completed (and all links could have been discovered).

Note that with debug log messages enabled, *POX* will continuously throw new update messages, but this is safe to ignore.

After *POX* and then *Mininet* have been set up, messages in the *POX* console will update on its discovery progress with messages that include ‘spanning tree updated’, ‘link detected’ while displaying links between nodes in the network, ‘ports changed’, and ‘requesting switch features’. Once these messages have stopped, it is safe to test the network.

The first test is to ensure the network can communicate, which can be found out by using the *‘pingall’* command in the *Mininet* console. It returns successfully with no dropped packets.

The second test is to see if *POX* detected the limited links and disabled those. To test this, use *Mininet*’s performance test between host 1 and host 3:

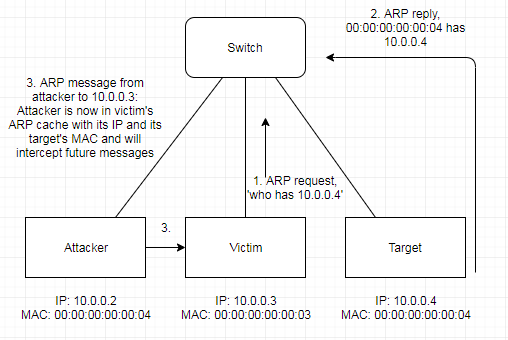
mininet> iperf h1 h3

This command reveals that the bandwidth between these two hosts averages 10 Mbit/s, which shows that *POX*’s spanning tree component does not look at bandwidth when disabling links. Further performance tests between other links (e.g. between hosts 2 and 4) also support this.

In conclusion, *POX*’s spanning tree component can create a spanning tree out of an OpenFlow network, but it cannot create a *Minimum Spanning Tree* [32] which would be the optimal way to work with a loop, weighted-graph network. Other controllers may support the MST algorithm, but *POX* would require a custom-made component that implements it for a *Mininet* topology.

*Experiment with Veriflow to detect ARP spoofing* - To showcase how *Veriflow* can be used to alert a user in the toolkit, this experiment will deploy a network in *Mininet*, where the *Mininet topology* contains a malicious host that will perform an ARP spoofing attack.

This topology consists of 3 hosts and a switch (see *Figure 7*). The malicious host has copied the MAC address of another host, meaning once it has sent a message to its victim, the victim’s now poisoned ARP cache will store the attacker’s IP with the target’s MAC. This means any future messages sent by the victim to the target will be intercepted by the attacker.



*Figure 7: A diagram of the topology, showing how ARP spoofing works.*

To set up the experiment, *Mininet* is executed through the toolkit with the custom topology ‘*ARP-Poisoning.py*’, and *Veriflow* with ‘*ARP-Poisoning.txt*’. Note that *Veriflow* must have access to the correct ARP cache, which is the ‘arp.txt’ file that is situated in the same directory as the *ARP Poisoning* topologies. This is because the experiment is simulating a scenario where a rogue host in the network has changed its MAC address to its target after the network is established, and in that real-world scenario, *Veriflow* would already know the attacker’s real MAC. For the sake of ease, the topology executes with the attacker having already obtained its target MAC. Performing this experiment with a typical SDN controller should be fine. For this report, the experiment will use *POX* and its *L2 learning switch* component [31].

The components are executed in this order; *POX, Veriflow, Mininet.* Follow the setup guide in *Appendix B* if there is any difficulty in executing any of the components.

Once the network is established, the experiment simply follows the steps outlined in *Figure X*. Commands are executed in *Mininet*:

mininet> h2 ping -c1 10.0.0.4

This sends a message to the host with IP 10.0.0.4, which is processed by the Switch, sent through *Veriflow* okay, is relayed to the controller, which creates corresponding flow entries for each host’s MAC addresses, and finally the ping is correctly sent and then replied in step 2. The victim (*h2*)’s ARP table now has the correct IP and MAC address of the target.

Step 3:

mininet> h1 ping -c1 h2

*h1* is the attacker. This message is intended to poison the victim’s ARP table, as it will create an entry with the attacker’s IP connected to the target’s MAC. However, *Veriflow* immediately detects the attempt, and throws a *‘detect arp poison’* exception. As this is a research copy, it does not do anything more such as stopping the network or attempting to fix the issue, but it does correctly identify the ARP spoofing attempt and flags it in its terminal.

Now, if the user attempts to perform another ping from the victim to the target:

mininet> h2 ping -c1 10.0.0.4

The ping hangs because the packet has instead been sent to the attacker, and the victim never receives a reply. This shows how *Veriflow* can protect a user from ARP spoofing, which is only one of the several network violations that it is designed to detect and flag.

# Conclusion

Creating the toolkit has been a great personal achievement. It is a fully functioning program that works with theoretically any OpenFlow controller through abstraction, has a network topology editor that can parse graphically made topologies into *Mininet* and *Veriflow* topology files and can remotely connect to a Linux VM through SSH terminals. It achieves all the goals that were established in the requirements section and is an application that could be used to efficiently experiment with OpenFlow networks.

Researching SDN and OpenFlow for this project was deeply interesting. Learning about the multitude of software that exists in the field, both the open-source software that is freely available to experiment with and the specialised business software that is used for large-scale projects shows how expansive the field is. SDN projects exist at every scale and range from general, simple software like *POX* to software for specific, but important roles such as *Veriflow*.

One conclusion regarding working with SDN in the future is that many of the applications that were researched for this project were very simple to understand and use in a network. Thanks to the nature of the SDN architecture [5], many SDN applications are designed in high-level languages with code that is easy to understand and, in some cases, customisable. Some open-source software such as *POX* are freely extendable with custom components, and some business software such as *Veriflow* [24] can be tailored for a business’s specific needs. This has made working with SDN a personally enjoyable and insightful experience and has been great preparation for potentially working in this field in the future.

There are some imperfections with the toolkit, some due to time constraints and some due to the language the toolkit was developed in. Developing the toolkit in Java was a safe bet. It is a comfortable language to work with and has good support for GUIs but does not have much networking library support. Finding libraries for specific purposes (e.g. interactive graphs, connecting to another machine via an SSH terminal, etc.) was challenging and cost a lot of time out of the project’s development. As mentioned in the body, the topology graph editor had to be recreated twice because development had started with libraries that did not have enough support for interactive graphs or did not have enough documentation to understand the library in good time.

There are also some problems with SSH connectivity. The original hope was to be able to transfer files from the user’s PC to a VM with a dedicated library, but instead a workaround had to be used where the toolkit must open command prompt, use the location of PuTTY provided by the user, and open SFTP through command prompt. This means the user must have PuTTY installed to properly use the toolkit, which might not be a necessary dependency if it was developed on another language.

The library used for direct SSH connectivity also has some bugs. When the user has multiple SSH terminals open (e.g. one with a controller running and one with *Mininet* running), if the user closes one terminal, every terminal closes. This is a oversight by the library creator, but this library is the only one in *Java* that opens an interactable terminal when connecting to a VM. Additionally, when running *Mininet* on the terminal, if the user attempts to open a new terminal for a specific node in the network (with the command ‘*xterm <host/switch name>’*), nothing happens.

In conclusion, this toolkit could have been developed with better support in another language that is known for having many networking libraries, such as *Python* or *Perl.* Creating the toolkit in *Python* would also likely enhance the toolkit’s support of *Mininet,* which would mean it could have been possible to use libraries to parse the topology rather than creating manual parsing code that became more complex to manage as more features were added.However, researching which libraries should be used for developing components of the toolkit in *Python* could have been more time-consuming, so using *Python* to develop the toolkit would have been a better option if there wasn’t as strict a time constraint on the toolkit as there is.

Another issue related to time constraints is the number of features that the topology editor supports. It supports most of the important features needed to experiment with *Mininet* topologies, but there are many other parts of the *Mininet API* [13] that could not be implemented into the toolkit as programming all of them into the toolkit would take too long. With more time and/or manpower it could be possible, but nonetheless most of the key features have been implemented.

The OpenFlow toolkit ultimately succeeds in the objective of acting as an abstraction layer that lets users import their own controllers, create their own topologies and experiment with their own simulated OpenFlow networks. Being able to specialise by implementing *Veriflow* and performing experiments with it was a great achievement that boosts the toolkit’s appeal. It stands as a great tool to ease its target group into the SDN field by giving newcomers a focused approach to creating and experimenting with OpenFlow networks and software of their choice, while also showcasing several flavours of the field through the topology editor and *Veriflow* to demonstrate what is possible with SDN. Creating it has given insight and real-world knowledge and experience into how businesses could use SDN to customise and experiment with their own networks.

# References

[1] McGillicuddy, S. (2013, April 17) Microsoft uses OpenFlow SDN for network monitoring and analysis. [*https://searchnetworking.techtarget.com/news/2240181908/Microsoft-uses-OpenFlow-SDN-for-network-monitoring-and-analysis*](https://searchnetworking.techtarget.com/news/2240181908/Microsoft-uses-OpenFlow-SDN-for-network-monitoring-and-analysis)*.*

[2] Matsumoto, C. (2016, October 14) Why Google’s Network Needs Five Controllers. [*https://www.sdxcentral.com/articles/news/googles-network-needs-five-controllers/2016/10/*](https://www.sdxcentral.com/articles/news/googles-network-needs-five-controllers/2016/10/)*.*

[3] Khurshid, A., Zhou, W., Caesar, M., & Godfrey, P. (2012, August). Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks* (pp. 49-54). ACM.

[4] Code of conduct | Membership | BCS - The Chartered Institute for IT. [*https://www.bcs.org/category/6030*](javascript:openWebLink('https://www.bcs.org/category/6030'))*.*

[5] Software-Defined Networking (SDN) Definition – Open Networking Foundation (2019). [*https://www.opennetworking.org/sdn-definition/*](https://www.opennetworking.org/sdn-definition/)*.*

[6] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., ... & Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, *38*(2), 69-74.

[7] Pfaff, B., Heller, B., Talayco, D., Erickson, D., Gibb, G., Appenzeller, G., ... & Kobayashi, M. (2009). OpenFlow Switch Specification. *2013-04—30].* [*https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf*](https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf).

[8] The POX network software platform. [*https://github.com/noxrepo/pox*](https://github.com/noxrepo/pox)*.*

[9] Code of conduct | Membership | BCS - The Chartered Institute for IT. [*https://www.bcs.org/category/6030*](javascript:openWebLink('https://www.bcs.org/category/6030'))*.*

[10] Mininet. [*http://mininet.org/overview/*](http://mininet.org/overview/)*.*

[11] Savage, M. (2017, May 21) SDN In the Enterprise. [*https://www.networkcomputing.com/data-centers/sdn-enterprise*](https://www.networkcomputing.com/data-centers/sdn-enterprise)*.*

[12] Cascado, M. List of OpenFlow Software Projects. [*http://yuba.stanford.edu/~casado/of-sw.html*](http://yuba.stanford.edu/~casado/of-sw.html)*.*

[13] Mininet Python API Reference Manual. [*http://mininet.org/api/annotated.html*](http://mininet.org/api/annotated.html).

[14] Star Topology – Telecom ABC (2005). [*http://www.telecomabc.com/s/star.html*](http://www.telecomabc.com/s/star.html)*.*

[15] Download PuTTY – a free SSH and telnet client for Windows. [*https://www.putty.org/*](https://www.putty.org/)*.*

[16] Apache NetBeans. [*https://netbeans.org/*](https://netbeans.org/)*.*

[17] java.com. [*https://www.java.com/en/*](https://www.java.com/en/).

[18] Trail: Creating a GUI with JFC/Swing (The Java™ Tutorials). [*https://docs.oracle.com/javase/tutorial/uiswing/index.html*](https://docs.oracle.com/javase/tutorial/uiswing/index.html)*.*

[19] JGraph/JGraphX. [*https://github.com/jgraph/jgraphx*](https://github.com/jgraph/jgraphx)*.*

[20] GraphStream – A Dynamic Graph Library. [*http://graphstream-project.org/*](http://graphstream-project.org/).

[21] JUNG – Java Universal Network/Graph Framework. [*http://jung.sourceforge.net/*](http://jung.sourceforge.net/)*.*

[22] sshd(8) – OpenSSH SSH daemon – Linux man page (2013, April 14). [*https://linux.die.net/man/8/sshd*](https://linux.die.net/man/8/sshd)*.*

[23] JCraft, Inc. JSch – Java Secure Channel. [*http://www.jcraft.com/jsch/*](http://www.jcraft.com/jsch/)*.*

[24] Network Verification Tools | Veriflow. [*https://www.veriflow.net/*](https://www.veriflow.net/)*.*

[25] The Board of Trustees of the University of Illinois (2012-2013). xwu64/Network-Monitor-Middleware, GitHub. [*https://github.com/xwu64/Network-Monitor-Middleware*](https://github.com/xwu64/Network-Monitor-Middleware)*.*

[26] Oracle (2018). Inet4Address (Java Platform SE 7).[*https://docs.oracle.com/javase/7/docs/api/java/net/Inet4Address.html*](https://docs.oracle.com/javase/7/docs/api/java/net/Inet4Address.html).

[27] Oracle (2019). java.util.regex (Java Platform SE 8). [*https://docs.oracle.com/javase/8/docs/api/index.html?java/util/regex/package-summary.html*](https://docs.oracle.com/javase/8/docs/api/index.html?java/util/regex/package-summary.html)*.*

[28] Canonical Ltd. (2019). Ubuntu Manpage: ovs-controller – simple OpenFlow controller reference implementation. [*http://manpages.ubuntu.com/manpages/trusty/man8/ovs-controller.8.html*](http://manpages.ubuntu.com/manpages/trusty/man8/ovs-controller.8.html)*.*

[29] Gentoo Foundation, Inc. dos2unix – Gentoo Wiki. [*https://wiki.gentoo.org/wiki/Dos2unix*](https://wiki.gentoo.org/wiki/Dos2unix)*.*

[30] tango-controls/JSSHTerminal: A pure java SSH terminal emulator (xterm). [*https://github.com/tango-controls/JSSHTerminal*](https://github.com/tango-controls/JSSHTerminal)*.*

[31] POX Wiki – Open Networking Lab – Confluence.[*https://openflow.stanford.edu/display/ONL/POX+Wiki.html*](https://openflow.stanford.edu/display/ONL/POX+Wiki.html)*.*

[32] Sedgewick, R. & Wayne, K. (2018, February 23). Algorithms, 4th Edition. Minimum Spanning Trees (Section 4.3). [*https://algs4.cs.princeton.edu/43mst/*](https://algs4.cs.princeton.edu/43mst/)*.*

# Appendices

## Appendix A – Project Log

This log will detail all the meetings between the project designer and the technical supervisor and reflect on the phases of the project that have been completed up to each point.

13/09/2018 – Initial Skype call. This mostly comprised of introductions and a simple explanation of this project, going into some examples of what the project would include and what would be expected.

18/09/2018 – First physical meeting. The focus of this meeting was to decide which final year project the student will undertake. To help, several research papers were sent via E-Mail after the meeting.

28/09/2018 – First official meeting for the project. From then on, meetings are to be fortnightly during the term. The meeting overviews the project, going into some details about the parts of SDN and information is sent containing the original OpenFlow research paper [1], several controllers and papers on Verification and SDN. Task 1 (Studying SDN Concepts) is started.

12/10/2018 – A talk about the project proposal, detailing what it will include. After that is a talk about the progress made so far. At this point the OpenFlow tutorial had been completed and it was possible to set up a simple OpenFlow network in *Mininet* with a controller. The next step is to explore further. An E-Mail was sent with a paper on *P4*, and the goal is to investigate implementing *Veriflow* into a network and to continue experimenting.

26/10/2018 – A talk about the interim report and what it should include. The immediate focus should be on reading more research papers to gain a stronger understanding of different facets of SDN to talk about in the report. There was also some theorising on what the project could consist of, e.g. a graphical interface that allows the user to click on buttons and have different networks be immediately set up, showcasing different parts of SDN. Ultimately, more experimenting is needed to decide where the project should go.

09/11/2018 – Discussion about the interim report, what is missing and what can be improved. Afterwards, discussion about creating the toolkit, emphasis on what it should include, e.g. an interactive graph where a user can make a network and then deploy it.

23/11/2018 – More discussion about the toolkit. Discussed the possibility of adding *Veriflow* to the toolkit. The decision is to start working on the toolkit now while also continue to research other parts of SDN, and then add *Veriflow* if it is still an interesting idea at that point.

07/02/2019 – First meeting of the spring term. Discussed how the toolkit is progressing. There were some setbacks as the graph component had to be recreated twice. Meetings will mostly be weekly from now on.

15/02/2019 – Online meeting to showcase progress on the toolkit. The graph component is mostly complete to a basic standard, next is to parse the graph into a topology file.

28/02/2019 – The graph section of the toolkit is complete by this point, next is to add the network capabilities of the toolkit so it can connect to a VM.

08/03/2019 – Online meeting. There was some discussion about adding unidirectional links to the graph editor, but it was concluded that it was not worth adding this as it is a minor feature. The focus is still on completing the network capabilities of the toolkit.

21/03/2019 – Showcase of the base version of the toolkit. The toolkit was transferred to a laptop which caused some issues in the presentation, but at this point it can now connect to a VM and transfer a controller and topology and can SSH into the VM with terminals. There was discussion on implementing *Veriflow* next. There was also some discussion on the poster and on writing the report.

29/03/2019 – Online meeting to properly showcase the toolkit.

04/04/2019 – Further discussion on implementing *Veriflow*. The issue is that it is a research copy and it is whether the time invested to get it working is worth the effort, as it may take too long and distract from writing the report.

09/04/2019 – Online meeting to discuss some of the issues about why *Veriflow* is not working.

11/04/2019 – Further discussion on *Veriflow*, but the focus is on writing the report. A deadline is set for 26/04 on sending a draft.

27/04/2019 – Online meeting to discuss feedback on the draft report.

There will be further meetings to discuss the final presentation at some point.

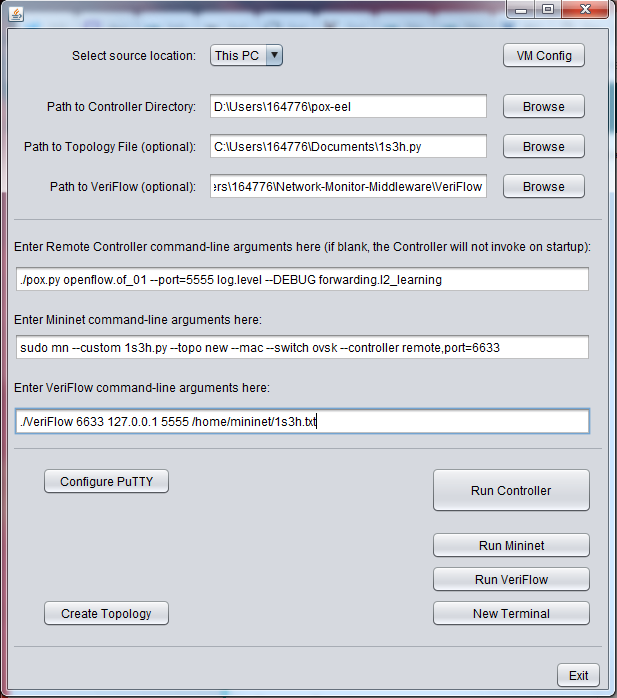
## Appendix B – Setup Guide

The OpenFlow Toolkit is a *NetBeans* project, which means it can be imported into *NetBeans* and ran through the main method in the *MainGUI* class there, or it can be ran directly through the .jar file in the *dist* folder in *OFToolkit*.

Please note that per the submission guidelines, the libraries that the toolkit uses are NOT included in the Canvas submission. However, they are on the GitHub page here:

<https://github.com/CharlieM1997/OFToolkit>

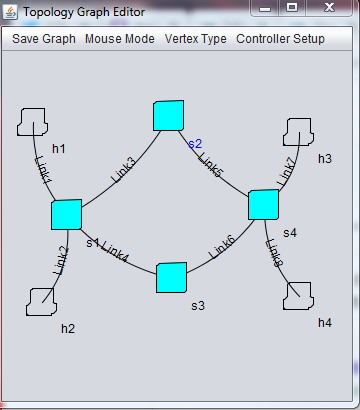
When the toolkit is opened, it will look like this:



There are several sections of the toolkit. First, go to *VM Config* and input your VM’s settings, such as its username, password and the IP address and port. Reminder that the toolkit has been tested with the *Mininet VM* located here: [10] Please also set the location of *PuTTY* on your PC. Try to give an absolute link if possible, as the toolkit may not recognise symbolic links.

If your remote controller and *Veriflow* are located on your PC, please keep the source location to ‘This PC’ and use the browse buttons to find their folders. If they are located on your VM, please set the toggle to ‘Mininet’, and manually provide the absolute links to their folders (e.g. the absolute link to a *POX* controller could be /home/mininet/pox).

If you wish to create a custom topology, press the ‘Create Topology’ button. Use the Vertex Type menu to swap between hosts and switches, click and drag on a vertex to make a link that you can drop on another vertex. Right-click on a node or link to get to their settings, or to delete them. Use the Controller Setup menu to customise the controller.



Once you are finished, click the ‘Save Graph’ button. The ‘Save As…’ option allows you to save the topology as a *Mininet* topology. The ‘Veriflow’ option will save it as a *Veriflow* topology. Please note that you need both when using *Veriflow*. Please also note that when saving a *Veriflow* topology, it will need an ARP table of the topology, which *must* be saved in the *Veriflow* folder.

Once you have created a *Mininet* topology, use the browse option to locate the file ready for transfer.

Please follow the respective tutorials of the controller of your choice and *Veriflow* when setting the command-line arguments for those applications. Please use the copy of *Veriflow* that comes in the *Network-Monitor-Middleware* folder. If replicating the *Veriflow* experiment in this report, follow the comments in *ARP-Poisoning.py*.

Typically, you will run the applications in this order:

*Controller, Veriflow (optional), Mininet*

Once your command-line arguments are set, run the controller, wait for it to transfer and once it is running in the terminal, optionally run Veriflow and then run *Mininet* and if successful, you should be able to then run your own *Mininet* commands within the terminal.