ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Rowan Klöti

OpenFlow: A Security Analysis

Master Thesis MA-2012-20
October 15, 2012 to April 14, 2013

Tutor: Vasileios Kotronis
Co-Tutor: Paul Smith
Supervisor: Prof. Bernhard Plattner

Abstract

This report contains a security analysis of the OpenFlow 1.0 protocol and the FlowVisor extension using the STRIDE method, as well as a description of possible attack methods using attack trees and a textual description of such attacks. The feasability of the attacks is analysed, and a practical demonstration of a number of the described attacks is performed. Subsequently, there is a discussion of countermeasures and mitigations. A review of newer OpenFlow specifications is also included.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 OpenFlow

This thesis deals with the security implications of *OpenFlow*, a protocol which implements *software defined networking*. A typical OpenFlow network would contain one or more *switches* and one or more *controllers*. A switch performs layer 2 and 3 switching using a *flow table*, a set of rules known as *flow rules*. These contain patterns used to match packet headers, as well as actions to perform on packets. The flow rules are installed on the switch by the controller. The controller may install such flow rules of its own accord, or it may do so in response to notification by the switch of a packet that failed to match existing flow rules. In the context of software defined networking, the switch constitutes the *data plane*, while the controller constitutes the *control plane*.

1.2 Motivation

OpenFlow is increasingly being deployed in production systems, and not just in academic environments[1]. For instance, Google is currently in the process of deploying OpenFlow in its internal backbone network, the so-called *G-Scale* network[26]. This is a major international backbone network, not a simple test environment. It seems likely that the growth of OpenFlow will continue into the future, with multiple major vendors offering support in their products - for instance Hewlett Packard[27] or Juniper[29]. In the past, security has all too often been a secondary consideration. Given the potential of OpenFlow to change the way that networks are managed, it seems appropriate to look into the security implications of OpenFlow while the technology is still at a nascent stage, yet to become entrenched in large enterprise deployments.

1.3 The Task

In this thesis, a method will be selected for performing a security analysis on the OpenFlow protocol. An analysis will be undertaken, addressing the potential security issues in OpenFlow itself, as well as new security issues that arise from the usage of OpenFlow. It is primarily the OpenFlow 1.0 protocol that will be examined, although newer protocol versions may also be inspected, in case new features result in new security issues becoming relevant. Important OpenFlow extensions should also be examined. In addition, an empirical demonstration of some of the the security issues discovered will be attempted.

1.4 Related Work

At the current time, little published work has dealt with the security issues of OpenFlow. There is no formal security analysis of OpenFlow, as far as the author has been able to determine. A significant quantity of work deals with potential security *benefits* of OpenFlow. There is also some published work on security-related *extensions* of OpenFlow.

[1]This does not preclude academic environments from being production systems

1.4.1 Security Extensions

One paper on the topic of security in OpenFlow, specifically in scenarios where flow rules may be considered untrustworthy is *A Security Enforcement Kernel for OpenFlow Networks*[49], which introduces the software extension called FortNOX[48] to the NOX OpenFlow controller[45], which provides a security system for OpenFlow systems. FortNOX provides a role based system with three levels of access: When a new rule is inserted, and if it overlaps with an existing one, the level of authorisation of the rule-inserting application will decide whether this rule will take precedence or not. In case the new rule takes precedence, the old one will be removed. The rules that are inserted are digitally signed and rules lacking a signature are allocated the lowest privilege level.
The paper *Carving research slices out of your production networks with OpenFlow*[55] proposes FlowVisor, a system allowing multiple virtual networks to be built on top of an OpenFlow network. It sits between the switches and the controller. As the title suggests, its primary application is to allow experimental research networks to be run over physical production networks, without the research network interfering with the operation of the production network. FlowVisor ensures full isolation between the virtual networks, which are known as “slices”. See Chapter 7 for more information.
The paper *VeriFlow: Verifying Network-Wide Invariants in Real Time*[31] proposes VeriFlow, a system used to validate the forwarding behaviour of a software defined network (OpenFlow-based) in real time. It also sits between the switches and the controller. The aim is to eliminate errors, such as routing loops or black holes as well as potential access control violations.

1.4.2 Security Applications

Numerous papers describe security applications of OpenFlow. For instance, the paper *OpenFlow Random Host Mutation: Transparent Moving Target Defense using Software Defined Networking*[28] describes a technique, labelled by the authors as “OpenFlow Random Host Mutation”. The technique exploits OpenFlow to protect end systems from attacks by providing them with a virtual IP address, visible from the outside of the network, which is translated into the actual IP address by the OpenFlow controller. This virtual IP address is changed rapidly, thus the notion of moving target defence.
The paper *Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow*[10] describes an application of OpenFlow to the detection of DDoS attacks, making use of *Self Organising Maps* to classify traffic patterns. The authors emphasise that their method requires less resources than existing detection methods without foregoing accuracy.

1.5 Overview

The rest of the thesis is organised as follows: Chapter 2 describes the approach used to perform the security analysis, with several papers on the subject being reviewed. It contains a description (2.5) of the methodology used for the security analysis. Chapter 3 contains a security analysis of the OpenFlow 1.0 protocol. It includes a description of the OpenFlow protocol (3.2), a model of the OpenFlow system using data flow diagrams (3.3), an enumeration of the vulnerabilities predicted by the model (3.4) and an attack tree, including textual description and feasibility analysis (3.5). Chapter 4 contains an analysis of the changes introduced in newer versions of the OpenFlow specification, insofar as they are security relevant. Chapter 5 contains the results of an experimental test of the security issues discussed in Chapter 3, as well as details of the setup used to achieve these results. Chapter 6 contains a discussion of potential mitigations and countermeasures to attacks. Chapter 7 contains a security analysis of the FlowVisor extension. Chapter 8 contains a discussion of potential future directions that research in this area could take. Chapter 9 contains a summary of this thesis.

Chapter 2

Methodology

2.1 Introduction

In this section, published methodologies for performing a security analysis (with a particular emphasis on network security analysis) will be reviewed, as well as the development of attack trees (which are analogous to fault trees, widely used in engineering applications). It is worth noting that for security models, there are those which focus on the attacker (attack trees are amongst these) and those that focus on the system (including Microsoft's STRIDE methodology, described below).

2.2 The STRIDE Methodology

2.2.1 Uncover Security Design Flaws Using The STRIDE Approach

[25] discusses Microsoft's approach to security analysis. The paper begins by introducing Saltzer and Schoeder's design principles:

- Open design
- Fail-safe defaults
- Least privilege
- Economy of mechanism
- Separation of privileges
- Total mediation
- Least common mechanism
- Psychological acceptability

The article then goes on to introduce a set of security *properties*, then a set of security *threats* and attacks against which they protect the system - the name STRIDE is derived from the first letters of the attack types, as seen in Table 2.1.
The article then continues on to *Data Flow Diagrams* (DFDs), which are a graphical representation of the data flow in a program. The diagrams model *data flows*, *data stores*, *processes*, *interactors* and *trust boundaries*. Data flows represent, for instance, network connections, while data stores may represent a database table. Interactors represent data producers and consumers "outside" the system, including the end user, and trust boundaries separate differing levels of trust. Each of the components is vulnerable to attacks as described in Table 2.2.

Attack type	Security property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-repudiation
Information disclosure	Confidentiality
Denial of service	Availability
Elevation of privilege	Authorization

Table 2.1: Attack types and equivalent security properties

Attack type	Data flows	Data stores	Processes	Interactors
Spoofing	✗	✗	✓	✓
Tampering	✓	✓	✓	✗
Repudiation	✗	✗	✓	✓
Information disclosure	✓	✓	✓	✗
Denial of service	✓	✓	✓	✗
Elevation of privilege	✗	✗	✓	✗

Table 2.2: Vulnerability of various component types to different sorts of attacks

The article further describes attack patterns, which can be used to model certain attack types, such as SQL injection or buffer overflow attacks.

2.2.2 Checking Threat Modelling Data Flow Diagrams for Implementation Conformance and Security

[1] describes the use of data flow diagrams, as used in the STRIDE method. It describes and summarises the same model as [25]. The paper introduces the concept of an *as-designed* and an *as-built* model, with the latter being derived from the source code, and a mapping between the two (so-called *Reflexion Models*). The edges (representing data flows) can be either *convergent* (existing in both models), *divergent* (existing only in *as-built* model) or *absent* (existing only in *as-designed* model).

For each of several potential security risks (spoofing, tampering, information disclosure, denial of service and ownership), rules are presented to recognise threats and potentially mitigating factors as well as remedies. The paper also contains a very simple example of a security model (for *Minesweeper*), as well as a formalised description of a DFD as it could be implemented in an object oriented programming language.

2.2.3 Other Papers

- The paper "A formal design of secure information systems by using a Formal Secure Data Flow Diagram"[59] presents a derivative of data flow diagrams which allows security properties to be formalised.

- The paper "Threat Risk Modelling"[2] presents an alternative model of a dataflow diagram, which the authors title "Flowthing model", claiming that it remedies some shortcomings of the dataflow diagram.

2.3 Attack Trees

2.3.1 Threat Modelling Using Attack Trees

[51], published in 2008, describes the use of attack trees in the security review of MyProxy[43], which is a credential management system used in grid computer applications. Their usage of attack tree is canonical - the same model as described by Bruce Schneier - with the aid of a software product called SecurITree[3], which provides graphical display of the attack tree as well as mathematical modelling. The described method for constructing the attack tree is as follows:

1. Define the attack objective, which becomes the root node.
2. Recursively divide this objective into prerequisite objectives.
3. This can be continued to arbitrary detail, but in general the idea is to decompose the attack into elements that we can quantitatively analyse, e.g. how hard is it to break a 2048 bit RS key?
4. Once we have reached the leaf nodes, we can then assign them values, for instance cost, or difficulty of execution.
5. These values are propagated up the tree, allowing one to make various calculations based on the model.

It must be said that the aim of this project is to understand security risks involved in Open Flow, not necessarily to quantify them. The danger in quantification is, given that the values of the leaf nodes are so uncertain, that the assignment of values would suggest a degree of precision that the model simply cannot support. The paper itself notes that attack trees are a high level methodology. Finally, constructing the attack tree is not performed in a systematic fashion, therefore it must be considered as a way for *describing* security issues (but in a systematic way) rather than *finding* them.

2.3.2 A Structural Framework for Modelling Multi-Stage Network Attacks

[14] provides an extension of the attack tree model, dividing nodes into *top level*, *state level* and *event level* ranked from most general to most specific. Furthermore, the concept of *explicit* and *implicit* links is introduced. These allow situations to be modelled where the execution of one node enables another node, i.e. they model capabilities which are conditional on other attacks. The concept of *context sensitive nodes* is also introduced; this is so that we can model scenarios in which attacks only work in certain security contexts.

2.3.3 Security Protocol Testing Using Attack Trees

[39] presents an application of attack trees for the analysis of the now-obsolete Wireless Application Protocol (WAP). The paper includes guidelines on how to construct attack trees - the recommended method is to begin with the ultimate goal of attacking the system, with attacks on different security properties (confidentiality, integrity and availability) as the second tier (the root node is of course an OR node). The next tier is the mechanism exploited by the attacker, and the subsequent levels are the steps required in order to perform the exploit. The main part of the paper is dedicated to the creation of actual attacks from an attack tree using an abstract language which is then transformed into executable code, which also allows attack scripts to be readily adaptable to other languages.

2.3.4 Other Papers

- The paper "System level Security modelling using Attack trees"[30] presents various extensions to the attack tree model, mostly allowing for concurrency (e.g. priority AND gate, see above). Many of these concepts can be realised with just AND and OR gates, however, or are only useful in exceptional circumstances (e.g. a *k out of n* gate).

- The paper “Vulnerability Assessment of Cybersecurity for SCADA Systems Using Attack Trees”[63] contains a security analysis of SCADA in power distribution networks using attack trees. The paper quantifies vulnerabilities, with a particular emphasis on password policies, which are known to be a problem on SCADA systems. The paper also ranks vulnerabilities, with a view to prioritise the ones with the greatest impact.
- The paper “Towards an Enhanced Design Level Security”[19] presents an approach for generating state charts from attack trees, similar to [20].

2.4 State-based and Other Methodologies

2.4.1 Capability-Centric Attack Model for Network Security Analysis

[58], published in 2010, deals predominately with the question of how attacker *capability* can be modelled. The paper defines capability as a tuple of *source*, *destination* and *rights*. The capability can be a *network* or a *host* capability. Network capability consists of physical, link and network access[1]. Host capability consists of OS, service and data access. In a similar manner, vulnerabilities are described as (from most abstract to most concrete): A concept vulnerability, an instance vulnerability, an exposure vulnerability and an exploiting vulnerability. The first defines a set of conditions that must be met in order to compromise a system, the second, the appearance of these conditions on a particular system, the third, the capability of an attacker to exploit an instance vulnerability, and the fourth the actual execution of an attack against such a vulnerability. In general, the successful exploitation of a vulnerability produces new knowledge and capabilities which can be, and usually are, used for further attacks on the system. If we compare this to the attack tree model, here we are looking at capabilities as leaf nodes, the exploitation of which allows us to obtain the capabilities and/or knowledge of the parent nodes. This would continue until we achieve some final objective, which is represented by the root node.

2.4.2 Modelling Security Attacks with Statecharts

[20] describes another method for security analysis, making use of *state charts*. The authors describe this approach as complementary to the use of attack trees. The paper describes several alternate methods, including *attack nets* (based on Petri nets) and the extension of UML for security modelling. The state chart model is attack-centric, modelling attacks with states, state transitions and events, which trigger the state transitions. The model contains AND as well as OR gates (in common with attack trees), but also has *priority AND* gates, which allow a *sequence* of events to be modelled. The AND gates are modelled as sets of states, where each input has two states (either “on” or “off”) and there is a set of states which allows the met input states to be counted. The OR gates are similar, but of course only require one input state to be “on”. The priority AND gate is similar to the AND gate, but requires that the events occur in a given order. In summary, it may be said that this paper maps the semantics of an attack tree to those of a finite state machine, adding timing properties to the model. It allows a more low level view of an attack and furthermore enables the numerous tools for simulating finite state machines to be utilised.

2.4.3 A Decade of Model-Driven Security

[5] as well as a number of other papers[6, 7] describe the approach of *model driven security*. The general idea of the usage of a system model to verify security is very much applicable to this thesis. However, it is important to note that we have no interest in describing *policy* (that is up to the system administrator) as opposed to the method of *implementing* (via the installation of *flow rules*) policy, nor do we have any interest in automated generation of operational systems (in the form of program code, hardware designs or other functional components) from system models. The papers deal in part with *role based access control*, which is again a question of policy, rather than implementation.
Finally, the paper [6] describes the use of process modelling in a security context. This is an application of control flow modelling, as opposed to data flow modelling (2.3.1), and is comparable

[1] This is clearly based on the OSI layered network model.

with the approaches involving state diagrams (2.4.2). This form of modelling can not readily be applied to OpenFlow, however. Although OpenFlow defines an API, that is, a series of *messages* that can passed between the the switch and the controller, it does not require any sort of sequential behaviour: Most of the OpenFlow messages are inherently asynchronous in nature and there is no requirement that either the switch or the controller react to each other's messages at all. Moreover, any reaction that does occur will be based upon policy, which in a software defined network may be defined by the administrator, or by third party developers. This is in contrast to a protocol where the reaction of the parties to messages is defined in the specification itself, such as TCP[50].

2.4.4 Other Papers

- The paper "A Threat Model Driven Approach for Security Testing"[69] introduces an automated modelling system, which allows code to be instrumented and recompiled, then run with randomly generated test cases, permitting the automated generation of threat models based on UML sequence diagrams.
- The paper "An Attack Modelling Based on Hierarchical Colored Petri Nets"[72] presents another method for modelling attacks, based on coloured Petri nets. The Petri net method is more sophisticated than attack trees, representing concurrency better, but is also not as easy to understand.

2.5 Conclusion

A number of a approaches to security modelling have been reviewed here. In general, they approach the issue from one of two perspectives: Either they attempt to model the *attacks* on a system, or they attempt to model the *system itself*, with the intent of finding potential avenues of attack. The first approach is *attack-centric*, the second *system-centric.*
For this project, the system-centric approach will be used initially, as the OpenFlow protocol can be readily described with a data flow diagram (or several). This will allow a better view of potential risks and vulnerabilities to be obtained, from which a set of attack trees can be derived. This can be repeated for more complicated attack models, or more complicated setups (multiple controllers or switches, use of FlowVisor et al.). Therefore, as the next step, we will:

1. Segment the OpenFlow system into modules.
2. Model these modules with data flow diagrams.
3. Analyse these data flow diagrams for potential attacks.
4. Create attack trees based on the scenarios that we have described in our attack models.

Chapter 3

Analysis of the OpenFlow 1.0 Specification

3.1 Introduction

In this chapter, the OpenFlow 1.0 specification will be subjected to a security analysis according to the STRIDE methodology[25]. First, a description of the OpenFlow 1.0 specification will be presented, followed by a data flow diagram modelling the data flows inside the OpenFlow controller/switch system. This model will be analysed for vulnerabilities using the STRIDE method. Finally, these vulnerabilities and their exploitation will be examined with the use of attack trees.

3.2 Specification

3.2.1 Data Stores

The most noteworthy data store of the OpenFlow 1.0 specification is the *flow table*. Its entries contain three sets of fields: *Header fields*, *counters* and *actions*. The *header fields* are used for matching. They can each contain either a fixed value or a wild-card which will match any value. Optionally, for IP-based fields, subnet masking is permitted, so that entire networks can be matched. See Table 3.1 for more information.

Field	Layer	Description
Ingress Port	Physical	Port of origin for frame.
Ether source	Link	Source MAC address
Ether dst		Destination MAC address
Ether type		Network layer protocol
VLAN id		VLAN ID, if 802.1Q tag present
VLAN priority		VLAN priority, if 802.1Q tag present
IP src	Network	Source IP address
IP dst		Destination IP address
IP proto		Transport layer protocol (TCP/UDP)
IP ToS bits		Terms of Service field
TCP/UDP src port	Transport	Source port (or ICMP type)
TCP/UDP dst port		Destination port (or ICMP code)

Table 3.1: Supported header fields

The *counters* fields are presented in Table 3.2. There are four possible scopes for the counters: Per table (essentially equivalent to global scope), per flow, per port and per queue (if multiple queues are in use). The counters wrap around seamlessly upon overflow, without any notification to the controller. Most of the counters are 64 bit values.

Scope	Values	Size (bits)
Per Table	Active Entries	32
	Packet Lookups	64
	Packet Matches	64
Per Flow	Received Packets	64
	Received Bytes	64
	Duration (seconds)	32
	Duration (nanoseconds)	32
Per Port	Received Packets	64
	Transmitted Packets	64
	Received Bytes	64
	Transmitted Bytes	64
	Receive Drops	64
	Transmit Drops	64
	Receive Errors	64
	Transmit Errors	64
	Receive Frame	64
	Alignment Errors	64
	Receive Overrun Errors	64
	Receive CRC Errors	64
	Collisions	64
Per Queue	Transmit Packets	64
	Transmit Bytes	64
	Transmit Overrun Errors	64

Table 3.2: Supported counter fields

The *actions* fields describe what the switch will do with the packets that it receives. Supported actions are *forward*, *drop*, *enqueue* and *modify field*. If there is no matching entry, the packet will be forwarded to the controller. If there is a matching entry, but no action, the packet is dropped; it is also possible to have multiple actions, which will be performed in order.

Drop is the default action. *Enqueue* is similar in effect to *forward*, but instead of being sent immediately, the packet is put in a queue on the corresponding port, allowing QoS to be used. Targets for forwarding are any physical port, as well as all ports, the controller (including encapsulation), the switch's local network stack and the port from which the packet originated. Optionally, it is also possible to send the packet along the switch's normal forwarding process (if the switch is not a pure OpenFlow switch) or make use of a spanning tree to flood the packet. See Table 3.3 for more information.

Action	Required
Forward	✓
Drop	✓
Enqueue	✗
Modify field	✗

Table 3.3: Supported actions

It is possible to modify certain fields of the packet, as described in Table 3.4. The fields which are supported for modification are the same as those supported for header field matching, except for the ingress port. Field modification actions would normally precede a *forward* or *enqueue* action. Setting a VLAN ID or priority will add the necessary tag, if it is not already present. Stripping a VLAN tag will remove it from the packet. All of the other modification actions are performed in situ.

Layer	Action
Link	Set VLAN ID
	Set VLAN priority
	Strip VLAN tag
	Modify MAC source address
	Modify MAC destination address
Network	Modify IP source address
	Modify IP destination address
	Modify IP ToS bits
Transport	Modify transport source port
	Modify transport destination port

Table 3.4: Supported modify field actions

Entries on the flow table can also be marked with an *emergency bit*. In the event that contact with the controller is lost and cannot be re-established, *all* entries apart from those marked with such a bit are removed. When connection to the controller is re-established, these entries remain, but may be removed if desired. The switch also uses these emergency entries upon starting up before an initial connection to the controller can be established.
Apart from the flow table, the switch is also assumed to have a per-port input buffer and output queue. The input buffer is used to store packets before they can be processed, and also while a response is expected to be obtained from the controller, if necessary. The output queue is used to store packets before transmission and to implement multiple per-port queues, which can be used for prioritisation.

3.2.2 Data Flows and Processes

The data flows in the OpenFlow setup are the regular network traffic and the traffic through the *secure channel* between the switch and the controller. The secure channel supports three types of communication: *Controller-to-switch*, *asymmetric* and *symmetric*.
The *controller-to-switch* traffic is used by the controller to control the switch. It is used during setup, to negotiate the supported features, to query and set switch configuration, to add, modify and remove flow rules as well as to query statistics. It is also possible to send a packet from the switch like this[1]. Finally, so-called *barrier* messages are supported. These allow packets to be

[1]Essentially allowing the controller to send packets from switch ports.

handled in a defined order: All packets received prior to the barrier message are guaranteed to be processed prior to any received afterwards.
The *asymmetric* messages cover the forwarding of messages to the controller - normally only the first 128 bytes in a *packet in* message[2] - the removal of flow rules (even when this has been initiated by the controller), changes in port status and errors. Packets whose headers are sent to the controller are accompanied with a buffer ID, which can be used to transmit the stored packet from the switch's input buffer.
The *symmetric* messages include *hello* messages, for connection instantiation, *echo* messages to determine liveness as well as the performance of the control connection, and finally vendor specific messages, which allow extensions to OpenFlow to be implemented.

3.3 Data Flow Model

3.3.1 Summary

Here, we model a simple scenario, namely the processing of packets, which may or may not create a new entry in the flow table. We assume OpenFlow 1.0 with no extensions on a router which is OpenFlow-only (no fallback to conventional switching). The model consists of two *clients*, independent from the system, identical and not trusted, a *switch* and a *controller*. The clients are each connected to an interface of the switch, while the controller is connected to a management interface.
Input packets to the switch are buffered, then processed by the switch's *data path*: Depending on the flow table, they may be *forwarded* or *enqueued* to an interface-specific queue before being sent through the attached interface, *modified*, sent to the controller or *dropped*. Packets with no flow table entry are forwarded to the controller, those with an entry but no action are dropped. Packets associated with multiple actions have all actions carried out in order. Modified packets are modified in-place (in the input buffer of the interface on which they were received), and then are subject to further actions, as specified by the flow table. It is possible to *enqueue* packets instead of *forwarding* them, allowing quality of service control to be used. The flow table contains a set of counters per-flow, as well as per-port and per-table. These are updated on execution of an action, and can be obtained by the controller. The controller may also have a packet sent out of an interface and install new flow rules.
The controller communicates with the switch via a network interface. It receives *asynchronous* messages from the switch, notifying it of new flows. It may react by installing a flow rule, or just having the packet forwarded, depending on a *policy*. The controller keeps a log, which may contain new flows established, as well as counter values that have been obtained from the switch. The log may be read, and the policy altered, by a system administrator, who may interact with the controller through an *administrative interface*. Note that although these components of the controller may be considered typical, they are not part of the OpenFlow specification, in which the controller can be considered a black box. Therefore, specific security issues arising inside the controller will not be given any further consideration. For more information about the switch implementation, see [41]. Please see A.1 on page 82 for more information on how data flow diagrams are to be interpreted.

3.3.2 Interactors

Client *n* This represents a client, considered external to the system and not trustworthy.

Administrator The system administrator, who may interact with the controller.

3.3.3 Data Stores

Input buffer *n* Where incoming packets are stored before and during processing. This buffer is specific to interface *n*.

Output buffer *n* Where outgoing packets are stored after forwarding to client *n*, if the *enqueue* or *forward* action is used.

[2]However, if the buffer is full, the *entire* packet should be sent.

Flow table This is the switch's flow table, as described in the OpenFlow specification. It has columns for matching layer 2, 3 and 4 headers, one or more actions and several counters.

Policy This models the rules by which the controller will decide how to react to a new flow. It may be altered only by an administrator.

Log This represents a log maintained on the controller.

3.3.4 Processes

Switch This process models the switch as a whole.

Controller This process models the controller as a whole.

Interface *n* This models the network interface connected to client *n*.

Data path This represents the data path of the controller. This is essentially the part of a switch that is implemented in hardware[3], and is independent of the operation of OpenFlow.

OpenFlow Module This represents the implementation-dependent OpenFlow software[4] on the switch, which is responsible for managing the flow table, as well as performing actions which are not supported by the data path.

Secure Channel This process models the network interface of the switch facing the controller, including SSL encryption, if it is in use.

OpenFlow Interface This process models standardised part of the controller software and the interface of the controller with the switch, including SSL encryption.

Decision This represents the "user-implemented" part of the OpenFlow controller. It is responsible for applying and enforcing forwarding policy and installing and removing flow rules as needed.

Administration Interface This represents an interface (e.g. web interface, CLI) via which an administrator may interact with the controller.

3.3.5 Data Flows

Sent packet to *n* This represents the transmission of a packet to client *n*.

Received packet from *n* This represents the reception of a packet from client *n*.

Controller-to-switch message This represents the sending of a *controller-to-switch* message, as defined in Appendix A of the OpenFlow specification, from the controller to the switch in order to send a packet, add a new flow table entry or query counters.

Asynchronous message This represents the sending of an *asynchronous* message, as defined in Appendix A of the OpenFlow specification, from the switch to the controller in order to notify of a received packet, port changes or errors[5].

Received packet A packet that is received from a network interface and copied into the input buffer.

Transmitted packet A packet that is forwarded from a queue, as a result of the *enqueue* or the *forward* action.

Forwarded/Enqueued packet A packet that is stored in a per-interface queue, as a result of the *enqueue* or the *forward* action.

Packet to process A packet which is read from the input buffer in order to be processed by the data path.

[3]This does not apply to a switch implemented purely in software.

[4]Here *software* also includes firmware.

[5]Note that this model does not include *synchronous* messages, also defined in Appendix A of the OpenFlow specification.

Remove/modify packet The packet remains in the input buffer until it is forwarded or dropped. The *modify* action results in the packet being changed in-place in the input buffer.

Transmit packet The controller can send a packet directly through the data path[6].

Packet sample If a new flow is encountered, a sample of the packet is sent to the controller to allow it to decide how to react. Normally, only the header is sent (by default 128 bytes), although in some circumstances the entire packet may be sent[7]. This data flow also represents the transition between the data plane and the control plane of the switch, as operations not supported by the data path will result in packets being forwarded to the software.

Read flow table Entries from the flow table are used by the data path, and can also be read out by the controller via the OpenFlow software.

Update counter The data path will update the counters in the matching flow table entry automatically.

Modify flow table New entries in the flow table are installed, existing entries are modified or old entries are removed in response to a flow modification message[8] generated by the controller. The OpenFlow software removes entries by itself when they reach their *hard*[9] or *soft*[10] timeout.

Get state/event The controller can query the state of the switch (e.g. the flow table) and is informed of events, such as the start of a new flow[11].

Set state/action The controller can modify the state of the switch (e.g. the flow table) and it performs actions, such as sending a packet[12].

Write log The logging process should regularly store logged events in a logfile.

Read log It is assumed that it is possible for the administrator to inspect the log via an administrative interface.

Read policy/write policy The administrative interface should also allow the forwarding policy of the controller to be modified.

Get policy In order to decide how to react to a packet received, the policy of the controller should be read first.

Get/set configuration The administrative interface should be able to set the configuration of the switch via the OpenFlow interface.

Set value/get value This represents the administrator's interaction with the administrative interface.

3.3.6 Boundaries

The system contains both *machine boundaries* and *trust boundaries.* The machine boundaries are placed between the clients and the switch, as well as between the switch and the controller. The only trust boundary runs between the data path (which belongs to the untrusted part of the switch) and the OpenFlow Module and flow table (belonging to the trusted part). The data path clearly deals with untrusted data, and is not capable of writing to the flow table, except for updating counters. The flow table is clearly trusted, as it controls the operation of the switch. By extension, the OpenFlow Module, which manages the flow table, must also be trusted.

[6] With the use of an *ofp_packet_out* message.
[7] This is the case if the input buffer is full and it is unable to store any new packets.
[8] That is, *ofp_flow_mod*.
[9] The duration after which the flow rule will be removed in all cases.
[10] The maximum idle duration permitted before the flow rule is removed.
[11] With the *ofp_packet_in* message.
[12] With the *ofp_packet_out* message.

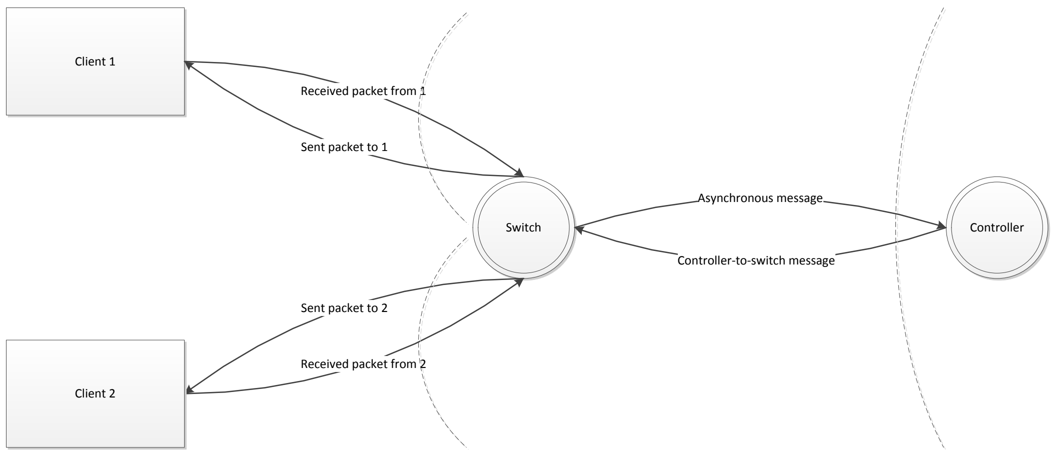


Figure 3.1: Data flow diagram of system

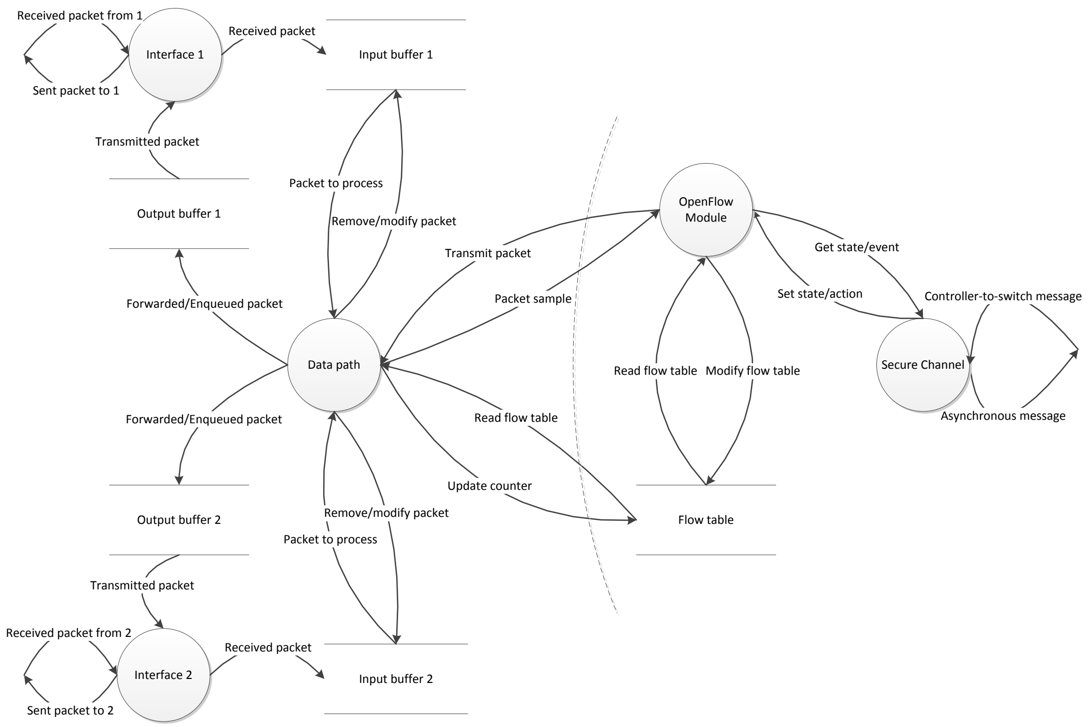


Figure 3.2: Data flow diagram of switch

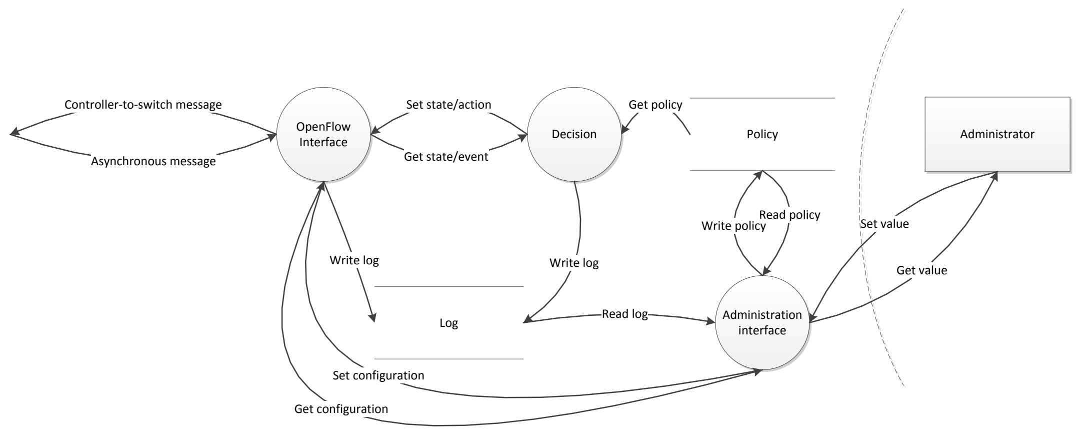


Figure 3.3: Data flow diagram of controller

3.4 Vulnerabilities

Given the model presented in 3.3, we can derive a list of *potential* vulnerabilities using the STRIDE method[25]. For each element, certain security issues must be considered - see 2.2.1 for more information on how this method is to be applied. Not all elements will be considered here. In particular, *data flows* which do not cross a trust or machine boundary will not be considered, as they generally represent internal data flows in hardware, or else function calls in software, and cannot be directly exploited. The *interactors* are considered to be external to the system and are therefore not given any further examination. Only components of the controller which could be subject to attack[13] will be examined. Exploitation of potential vulnerabilities will be discussed in 3.5.

3.4.1 Data Stores

According to section 2.2.1, we must consider the risks of *tampering*, *information disclosure* and *denial of service*. The risk of *repudiation* is also relevant, but does not apply here, apart from log files.

3.4.1.1 Input Buffer

The input buffer receives untrusted data - its contents are not considered trustworthy. There is no way to modify the existing contents of the buffer by sending packets externally. Therefore, the risk of *tampering* does not apply here. There is a small risk of *information disclosure* - it is possible that a side channel attack may disclose the capacity and/or the current load of the buffer. There is a much more significant risk of *denial of service* - The packets remain in the buffer until forwarded or dropped. If it is necessary to send a packet header to the controller, the packet concerned will remain in the buffer until a reply is received. An overflow of the input buffer would also have dramatic effects on other system components, as the specification requires full packets to be transmitted if existing ones cannot be saved in the switch buffer.

[13]i.e. that are “reachable” from untrusted data sources.

3.4.1.2 Output Buffer

The output buffer cannot be considered vulnerable to *tampering* for the same reasons as above. *Denial of service* is not relevant, as the output buffer is not likely to be the limiting factor in any attack. It is unlikely that any *information disclosure* attack (i.e. side channel attack) would reveal anything for the same reason.

3.4.1.3 Flow Table

The flow table is the most important data store, being the component which is specific to OpenFlow. *Denial of service* is a significant threat, as the capacity of the flow table is limited, and each flow rule contributes to filling that capacity. Depending on how the internals of the flow table are implemented, more sophisticated attacks are conceivable. *Tampering* is not directly possible, as the entries of the flow table are generated by the controller, however, it may be vulnerable to attacks that involve the controller. *Information disclosure* is conceivable - it is possible to imagine an attack that fills the flow table in order to discover its capacity. More interesting attacks are described in 3.4.3.1 and 3.4.3.2.

3.4.1.4 Policy

This component is not vulnerable to *tampering*, as policy will normally not be changed in response to user network traffic. It is also not vulnerable to *denial of service*: From the switch's perspective, it is read-only. On the controller, any performance issues would be covered by the process *Decision*. *Information disclosure* is possible through a side channel attack, which is considered an attack against the relevant process and discussed in 3.4.2.6.

3.4.1.5 Log

The log may be considered to be either vulnerable to *denial of service*, if new entries are always appended, or *tampering* if old entries are overwritten[14]. *Repudiation* is not an issue, as packets received are not trusted. *Information disclosure* is not relevant, as there is no way to read the log from the switch, and logging can be considered to be asynchronous[15].

3.4.2 Processes

Processes are subject to all types of security issues covered by the STRIDE methodology. However, not all of these types are relevant to this particular application. *Elevation of privilege* issues are not an issue, as there is no code being run in response to user-generated data and no concept of different levels of privilege. Such a security issue may become relevant in other scenarios, such as the use of FortNOX[49]. *Spoofing* is not an issue (although it may also become relevant for FortNOX): IP and MAC addresses from packets can obviously be spoofed, but this does not have adverse effects on OpenFlow above and beyond normal networks. However, *tampering* may be an issue if spoofing is in use, due to the possibility that the controller installs flow rules covering multiple flows based on untrusted data (user supplied packets). *Repudiation* is not an issue for the same reason as spoofing: There is no concept of authenticity in IP networks[16]. *Denial of service* and *information disclosure* are both possible.

3.4.2.1 Interface *n*

The network interface is implemented in hardware and is designed to operate at line rate. Moreover, it functions no differently from the network interface of a conventional switch. For this reason, no security issues arise here.

3.4.2.2 Data Path

The data path is the part of the switch that is implemented in hardware. As this part of the switch is designed to operate at line rate, it should not be considered vulnerable to *denial of*

[14]This could be the case if the log is written in a circular pattern, with the latest entry replacing the oldest.
[15]The controller does not wait for the logging to be completed before sending a command to the switch.
[16]Excluding the use of IPSec, but this is enforced by endpoints.

service attacks. If the switch is incapable of processing packets at line rate, then its design must be considered flawed. As the data path is assumed to process packets at the same speed independently of their contents, *information disclosure* is not possible. *Tampering* is not possible, as the data path simply follows the instructions in the flow table. Any attack would therefore be based on flow rules being installed in the flow table by the controller.

3.4.2.3 OpenFlow Module

This is the part of the switch that is implemented in software. Insofar as the data path of the switch does not support OpenFlow operations, these must be implemented on the switch software instead. This may result in *denial of service* attacks. Side channel attacks based on such performance differences are described in 3.4.3.1. *Tampering* attacks are not possible here for the same reason as 3.4.2.2.

3.4.2.4 Secure Channel

The secure channel operates an encrypted channel over SSL. This requires a certain amount of processing power, unless the encryption is performed in hardware. If it is possible to obtain access to the management network, it is also possible to attack the interface with a variety of conventional TCP/IP attacks. This component is therefore vulnerable to *denial of service* attacks. Due to strong encryption being used, we can be reasonably sure that *tampering* and *information disclosure* are not possible, but see also 3.4.3.3.

3.4.2.5 OpenFlow Interface

This component represents the counterpart to the secure channel (3.4.2.4) in the controller. It has the same vulnerabilities. It is likely that the controller has more processing power than the switch, but the likelihood of the controller having encryption performed in hardware is lower. The controller is more vulnerable to *denial of service* attacks if it controls multiple switches concurrently, which is not represented in the situation modelled here, but is the most likely scenario in practice.

3.4.2.6 Decision

The decision module is where the “user” of the system implements the control logic. This may be implemented using Python or another scripting language. At any rate, this logic may be vulnerable to *denial of service* attacks if new flows are generated at a high rate. A “smart” controller may implement flow rule *aggregation*, such that several flow rules are aggregated into one flow rule using wildcards and subnet masks, instead of simply reactively installing flow rules to match single flows. If such a system is in use, then the logic is also vulnerable to *tampering*, as user-generated, untrusted content is being used to install flow rules which have an effect on other users or systems. The logic may also be vulnerable to *information disclosure*, in order to determine whether flow rules are being installed or not. This is described in 3.4.3.3.

3.4.2.7 Administration Interface

The administration interface of the controller, assuming that it has one (a simple system might have its configuration hard coded), is a potential avenue for attacking the system. It is not part of the OpenFlow specification and therefore is not covered in this thesis, although any comprehensive audit of a working system must include all elements.

3.4.3 Data Flows

Only data flows that cross process, machine or trust boundaries will be considered here. The purpose of this is to reduce the number of irrelevant elements requiring consideration.

3.4.3.1 Packet Sample

This data flow represents the transition between the hardware and software components of the switch, assuming that the entire switch is not implemented in software. As the software components are less optimised for processing packets at line rates, operations which are not supported by the switch's data path are significantly slower. This represents a *denial of service* vulnerability (described in 3.4.2.3), but also an *information disclosure* vulnerability: It may be possible to exploit the timing differences to learn what operations a switch is performing on a packet. This will require accurate timing information and a large data set, unlike 3.4.3.3. *Tampering* is not a concern due to the fact that this is an internal data connection with no possibility of outside interference.

3.4.3.2 Update Counter

This data flow represents the special case where the flow table is written by the data path, namely to update counters. This represents a risk of *tampering*, as the counter values could be forced to overflow. There is no overflow notification in the OpenFlow 1.0 protocol, so this would falsify the values, although this is likely to be very difficult in practice due to the range of the counters[17]. It is also conceivable that, with the use of forged values in the packet header, counters of other flows may be influenced, which may constitute a security issue, depending on how they are used. There is no possibility of *denial of service* or *information disclosure*, as the update happens at line rate and an overflow does not impact performance or otherwise cause disruption.

3.4.3.3 Asynchronous Message

This data flow represents the actual transmission of data from the switch to the controller. It is unique to the OpenFlow setup and from a security perspective one of the most problematic elements. *Tampering* is not possible due to the use of SSL. *Denial of service* is possible: The rate of traffic to the controller is proportional to the number of packets that do not match existing flow rules. If the network capacity is exceeded and the network becomes congested, the switch's reaction to new flows will be adversely affected. Since there is a substantial time difference between packets processed locally on the switch and those that must be transmitted to the controller, it is possible to determine whether or not this transmission has taken place. This leads to *information disclosure*, making it possible to determine whether a packet matches a flow rule or not. With subsequent sending of additional packets, it may also be determined if a new flow rule is installed, as well as the duration of the flow rule. Additional issues arise if an attacker is assumed to have direct access to the management network. In this case, traffic analysis can be performed on the transmissions between the switch and the controller. This would reveal more information than can be obtained via timing analysis, as it is also possible to determine the size of any transmitted packets, as well as the controller's response. With access to the management network, it is also possible to use various conventional TCP/IP attacks to disrupt the switch-controller connection, resulting in additional possibilities for *denial of service* attacks.

3.5 Attack Trees

3.5.1 Introduction

Attack trees are a schematic method for representing and analysing security risks, based on fault tree analysis[71]. The visualisation here is also based on a fault tree, which can also be considered as a superset of an attack tree (although there are proposals to extend attack trees with some of these elements). For more information, see A.2 on page 83. All of the attack trees presented here are reproduced in a larger size in C on page 85 for better readability.
According to the STRIDE methodology, there are six potential attack types that must be addressed:

Spoofing Consists of attempting to fool the system that we are somebody other than who we actually are. It is possible to spoof a MAC address or an IP address; it is also possible to send forged ARP and/or IPv6 router advertisement packets to attempt to solicit traffic destined for other hosts. This is not a risk that is specific to OpenFlow, however.

Tampering Consists of falsifying information. If we can get the system to modify data that does not originate from us, than we have succeeded in performing such an attack. Possibilities

[17]Many are 64 bits long.

here are the falsification of flow counters, and tricking the controller into the installation of flow rules which may perform some packet modification.

Repudiation Consists of denying responsibility for content generated by us. As stated above, we may forge source addresses for packets, therefore there is no assumption that packets genuinely originate where the packet headers indicate.

Information disclosure Consists of obtaining information that we are not entitled to see. In this context, this applies mostly to side channel attacks to reveal more information about the OpenFlow system as a whole than would be available if a regular switch was being used.

Denial of service Consists of preventing the system from transmitting information normally. This is the security risk where OpenFlow provides the largest attack surface: The fact that packets must be sent to the controller on a regular basis opens up a number of potential avenues for denial of service attacks.

Elevation of privilege Consists of obtaining the ability to perform operations that we are not entitled to perform. The only potential way we could do this is by assuming control over the controller, which is assumed to be infeasible due to the use of SSL. This may become relevant when considering more complicated variants of OpenFlow systems, such as those where multiple authorities are permitted to inject flow rules into the system.

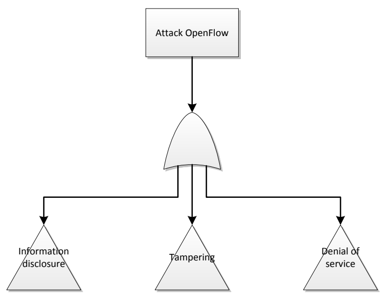


Figure 3.4: Overview attack tree

3.5.2 Tampering

Here we have the possibility of tampering with either the counters, or attempting to modify the behaviour of the switch. We may attempt to overflow the counters, although this appears to be relatively infeasible due to the length - most are 64 bit integers. We may also attempt to change the counters of *another flow*. We would do this by identifying a flow in the flow table (possibly guessing), then sending packets with the relevant header data - assuming that the flow rule does not restrict our choice of port. This may be relevant if the counters are to be used for billing purposes, which is certainly *not* to be recommended, unless the port numbers are restricted. Even then, it constitutes a security issue, as a denial of service attack against a host could be used to falsify costs as well as deny availability.
Another possibility depends on the routing policy - specifically, how general or specific the flow rules are. If relatively generic rules are used, we may attempt to fool the switch[18] regarding the identity of a system connected to a particular port. If we then establish a flow from another port, and the controller installs a reasonably generic flow rule, we may be able to capture traffic intended for a particular client. Even if the controller does not install generic rules, it may still be possible to fool it into installing a flow rule which would be used to divert traffic, as flow rules are not automatically removed upon transmission of a TCP FIN packet. Alternatively, if the controller installs a rule to modify packets, we may target this instead, also by sending forged packets. It is less clear what such an attack might accomplish, although it should be taken into

[18]For instance, with forged ARP packets or router advertisements, or forged routing protocol packets.

consideration if, for instance, the packet contents are being used to add a VLAN tag, or QoS data.

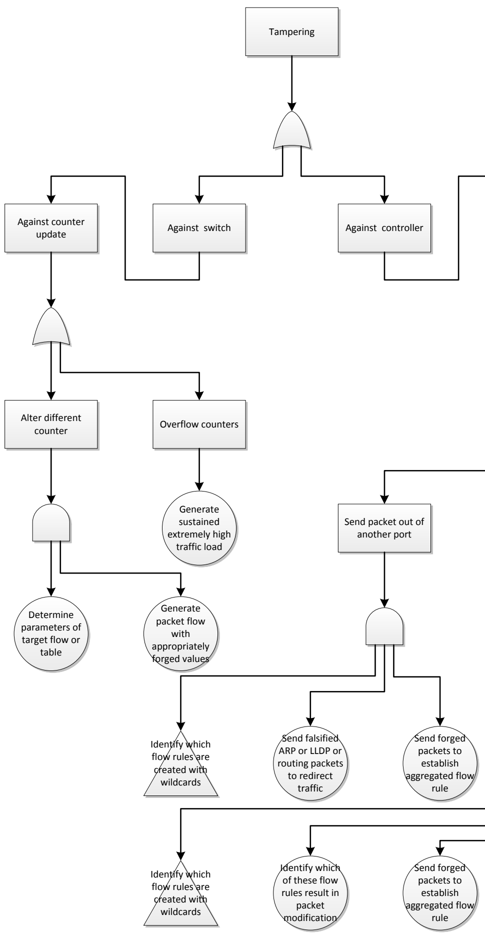

Tampering
Against counter update
Against switch
Against controller
Against Decision process
Attack flow aggregation
Alter different counter
Overflow counters
Generate sustained extremely high traffic load
Send packet out of another port
Modify packet
Determine parameters of target flow or table
Generate packet flow with appropriately forged values
Identify which flow rules are created with wildcards
Send falsified ARP or LLDP or routing packets to redirect traffic
Send forged packets to establish aggregated flow rule
Identify which flow rules are created with wildcards
Identify which of these flow rules result in packet modification
Send forged packets to establish aggregated flow rule

Figure 3.5: Tampering attack tree

3.5.2.1 Counters

The OpenFlow standard includes various counters for obtaining statistical information. This includes values per-flow, as well as per-port and per-table (i.e. globally). In the model, the updates to the flow table are represented by the data flow *Update Counter*. The OpenFlow standard does not require a notification in case of counter overflow, nor does it provide a method with which this could be done. As such, it would be straightforward, but very tedious, to attempt to cause these to overflow. It would be useful only if the counters are being used for purposes other than statistics. The formula

$$t_{overflow} = \frac{s_{pkt} 2^{n_{bits}}}{R_{clnt}} \tag{3.1}$$

where

$t_{overflow}$	is the time required to effect an overflow
R_{clnt}	is the throughput available between the client system performing the attack and the switch
s_{pkt}	is the size of packet required to match an existing flow rule and
n_{bits}	is the size of the counter field in bits

gives the amount of time a flow must be sustained before an overflow will occur. It may be observed that whereas this attack type would be marginally feasible for 32 bit fields, the relevant fields in the OpenFlow specification are 64 bit, resulting in an exceptionally long duration of time before an overflow occurs.
Apart from this, it would also be possible to influence counters by sending forged packets. Once again, this is useful only if, for example, the connections are being metered. A packet with forged source address could then be used to assign traffic to a flow intended for another user. If such a system were in use, it would more likely be based on switch port numbers rather than IP addresses that can be spoofed.

3.5.2.2 Redirecting Flows to Another Port

It must be said of this attack that it is not a weakness of OpenFlow per se, as it occurs on the controller (specifically, at the *Decision* process). Rather, it is a potential method to take advantage of *flow aggregation*. A purely reactive strategy is unlikely to be vulnerable to such an attack, but would be more vulnerable to other attacks[19]. If aggregation is being performed, the controller may coalesce several flow rules which "overlap" (same source and destination ports, same or neighbouring destination address(es), neighbouring source addresses) into a single rule. If the controller is making use of ARP, LLDP or some other link-layer discovery protocol that is vulnerable to spoofing, then a malicious system may spoof a desired destination system - a server of some sort would be the obvious target, as it is most likely to be connected to by network clients. An accomplice, connected via the same port as the victim system, would then itself create a variety of connections to the malicious server, using forged (but neighbouring) source addresses. The intent is that the controller should install a flow rule for the entire subnet to which the accomplice (and the victim) is connected to. This rule would then point at the malicious system, rather than the real server. Periodic traffic generated by such an accomplice could be used to ensure that the flow rule does not timeout. Alternatively, the server could solicit responses from several systems on the victim's subnet, which would unwittingly become accomplices.
Instead of ARP or LLDP, a malicious system could also spoof the MAC address of the target system. If the controller does not store a MAC address/port mapping, it may be able to take over traffic from a server with the same MAC address, with the controller believing that the server has been connected to a different port. This would require that flow rules pointing to the malicious system expire first. Then the malicious system, with the MAC address of the target system, would generate traffic as if it had been newly connected. If an attempt is made to connect to the system, the controller may install a flow rule pointing to the malicious system and the switch port it is connected to.

3.5.2.3 Having Packets Modified

This attack[20] is also an attack on the *Decision* process. Apart from having packets modified, an attacker may also want to have the packets in a flow modified. Clearly, this is only possible if the controller would install a flow rule which modifies packets[21] in response to user traffic. This would only constitute a security issue if the installation of the flow rule resulted in packet modification not intended by the controller policy.

3.5.3 Information Disclosure

Here we could attempt to find out various things about the switch or controller. We may try to find the contents of the flow table, such as which flow rules exist, what their actions are as well as attributed counter values. We may also target the controller to find out what the forwarding policy is, i.e. what flow rules will be installed in response to various input packets. Theoretically, we could also attack input and output buffers on the switch, but it is not clear how such an attack could be carried out.
If we have access to the management interface, we may attempt to perform a traffic analysis. If we are unable to intercept traffic, but yet can still connect to the interface, we may still be able

[19]See 3.5.4.1 for an example.
[20] A variant of 3.5.2.2
[21]This could be to strip VLAN tags or implement a NAT, for example.

to determine when information is being transmitted. For instance, by pinging the controller and comparing response times: A longer response time may indicate traffic on the network interface, assuming that the controller's TCP/IP stack processes the packets in order. This would be a *timing analysis* (side channel) attack. Either type of attack would attempt to determine whether a packet header is being sent to the switch, as well as whether the switch installs a new rule or not. Depending on how detailed our information is, we may be able to determine the quantity of data transmitted from the controller to the switch, allowing us to infer the number of actions that the flow rule contains.
In the more likely scenario where we do not have access to the management interface, it may still be possible to determine either the contents of the flow table or the controller's policy. Consider that there are fundamentally three possibilities for a packet received by the switch: Firstly, it may be forwarded entirely in hardware, which is very fast. It may also be sent to switch control plane, due to the fact that the action specified is not supported by the switch's data plane. This would take somewhat more time, although this is very much dependent on implementation. Finally, the switch may send the packet (or its header) on to the controller, which is much slower, and therefore easier to detect. By detecting the latter case, we can determine whether a flow rule exists. If we send the same packet again, we can also establish when a new flow rule is created. With a much larger number of packets, we may be able to differentiate between those which are handled entirely by the data plane, and those which are handled in the switch's control plane. This allows us to determine what actions the switch is taking on the packet, if we cannot determine this otherwise. These attacks require that we have access to at least two client interfaces, unless we can have a client reflect back our packets in a predictable way.

3.5.3.1 Determining whether a Flow Rule Exists

The objective of such an attack (on the *Asynchronous Message* data flow) would be to attempt to probe that status of a network - which hosts are active and what they are doing. The attack is straightforward to perform, utilising the fact that a packet matching a flow rule will be forwarded far more quickly than one without. The main difficulty is ensuring that the reply can be received. If flow rules are *aggregated*, it may be possible to send packets which will be covered by a flow rule set up for another client, which would cover flows with a set of source addresses. If this is not possible, forging a source address would be necessary. This would make it difficult to receive a reply. We may elect to use ARP or LLDP spoofing in order to associate ourselves with another IP address. If possible, sending forged routing packets may also be useful. Whether this will work depends on how the controller is configured - it may not accept the use of certain addresses for the port which is being used to communicate with the switch. There is also a danger of disrupting IP traffic to and from the system whose address we are targeting - this would constitute a denial of service attack in its own right[22], but here the emphasis is on stealth. It may be necessary to repeat the procedure in order to increase statistical certainty. We may also use repetitions with varying delays to attempt to find out what the duration of the flow rule is - invaluable information for a denial of service attack. In any case, and with such kinds of attacks, they should be performed at low intensity if they are to be repeated, in order not to trigger any alarms on firewalls or IDS systems.
Let X_{switch} be the random variable representing the processing time[23] for packets handled on the switch alone, and $X_{controller}$ be the random variable representing the processing time for packets which are forwarded to the controller, including any time required on the controller for a new flow rule to be created. Assume we have μ_{switch} and $\mu_{controller}$ as the mean values of X_{switch} and $X_{controller}$ respectively, and $f_{X_{switch}}(x)$ and $f_{X_{controller}}(x)$ as their probability distribution functions, which have the following lower bounds (as the processing time cannot be less than the transmission time):

$$\forall x : (x < \frac{s_{pkt}}{R_{clnt}})\ f_{X_{switch}}(x) = 0 \tag{3.2}$$

$$\forall x : (x < \frac{s_{pkt}}{R_{clnt}} + \frac{s_{pkt}}{R_{mgmt}})\ f_{X_{controller}}(x) = 0 \tag{3.3}$$

We need a threshold value t such that

[22]Not one that will be covered here, as conventional networks are vulnerable to this in equal measure.
[23]Processing time is understood to be round trip time and transmission time

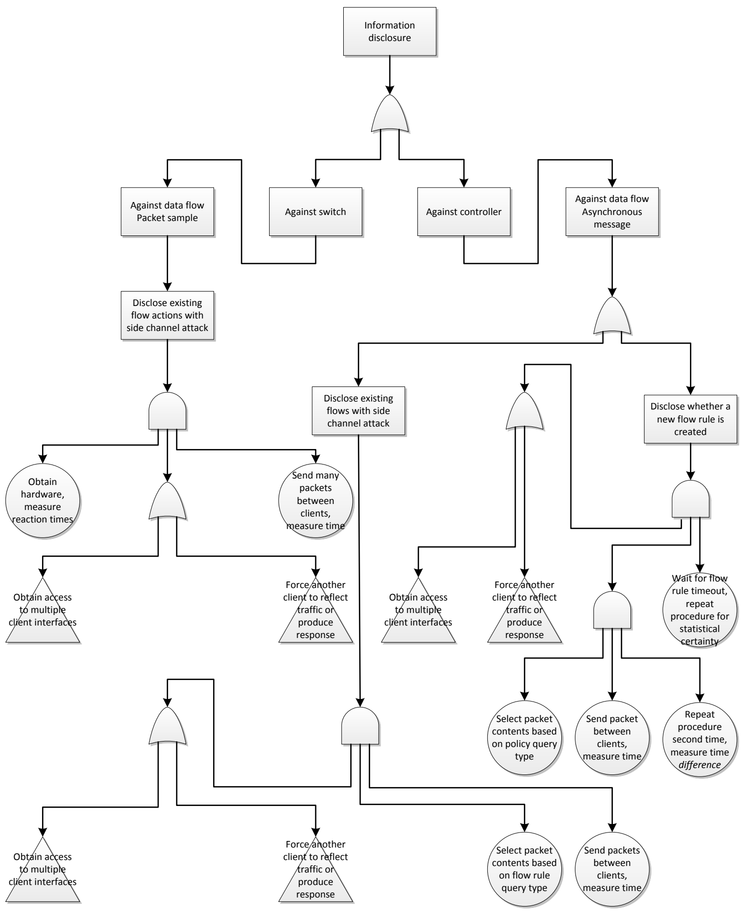

Information disclosure
Against data flow Packet sample
Against switch
Against controller
Against data flow Asynchronous message
Disclose existing flow actions with side channel attack
Disclose existing flows with side channel attack
Disclose whether a new flow rule is created
Obtain hardware, measure reaction times
Send many packets between clients, measure time
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Wait for flow rule timeout, repeat procedure for statistical certainty
Select packet contents based on policy query type
Send packet between clients, measure time
Repeat procedure second time, measure time *difference*
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Select packet contents based on flow rule query type
Send packets between clients, measure time

Figure 3.6: Information disclosure attack tree

$$\mu_{switch} < t < \mu_{controller} \tag{3.4}$$

If we have n sample times x_i, then we have

$$\overline{x} = \frac{1}{n}\sum_{i=0}^{n} x_i \tag{3.5}$$

For $\overline{x} < t$ the hypothesis is that the packet is forwarded by the switch directly, for $\overline{x} \geq t$ the hypothesis is that the packet is forwarded to the controller. We have errors e_1 and e_2 with

$$e_1 = \Pr(Packet\,handled\,in\,switch\,exclusively | x \geq t) = \int_t^{\infty} f_{X_{switch}}(x)dx \tag{3.6}$$

$$e_2 = \Pr(Packet\,forwarded\,to\,controller | x < t) = \int_0^{t} f_{X_{controller}}(x)dx \tag{3.7}$$

The objective is to find a t and a minimum n such that e_1 and e_2 do not exceed a given threshold. Increasing the value of t should decrease the value of e_1 at the expense of e_2, so we should pick t in the middle of the two, if the standard deviation of both distributions are equal. If the standard deviation of the distributions is not equal, then we should choose a threshold closer to the mean of the distribution with the smaller standard deviation. The following formula returns a threshold t, which is the linear combination of the means of both distributions, with each mean's contribution to the threshold value being proportional to the other distribution's standard deviation σ:

$$t = \mu_{switch} + (\mu_{controller} - \mu_{switch}) \cdot \frac{\sigma_{switch}}{\sigma_{switch} + \sigma_{controller}} \tag{3.8}$$

3.5.3.2 Determining what Action a Flow Rule Will Take

The objective of this attack would be to find out which actions a particular flow rule is configured with. The essential attack methodology is the same as above, but instead of attempting to determine whether a packet is sent to the controller, we attempt to determine if the switch is performing some extra processing. The premise is that the switch can either perform operations in hardware (which is the fastest), on its own processor (less quick), or it may send the packet to the controller - which is much slower. This is an attack against the *packet sample* data flow (3.4.3.1).

Assuming that we can obtain a sample device - which requires that we determine the device in use - and probe it extensively, we may be able to determine the timing characteristics of certain operations: Obviously, if the operation is to forward to the controller (this is one of the virtual ports available), this will be noticeable. But other virtual port forwarding operations may be implemented in software instead, and it is also possible that the packet modification operations are implemented in software too. Some of these operations (such as packet modification) are also directly observable. This would seem to make such a timing attack useless - however, we could consider the same in reverse, namely use the timing characteristics of the switch to determine the hardware in use. In order to obtain certainty, a large number of packets will have to be sent, as the time differences are on a much smaller scale of magnitude, compared to the round trip time of the connection. Operations which use the software of the switch may offer a possibility for a denial of service attack, especially if the operation involves sending a packet to the controller.

3.5.3.3 Determining whether a Flow Rule will be Created

The objective of this attack on the *Asynchronous Message* data flow is to find out whether a new flow rule is created in response to a particular packet. This is of particular interest if we wish to perform a denial of service attack, as here the switch will not be able to operate at line rate. The attacker would send two packets, with a short delay in between, allowing the controller to install a flow rule, if needed. If a new flow rule is created, the time duration between the replies to the packets will be much smaller for the second packet than for the first. This attack is useful even without any kind of source address spoofing, as we are less interested in the current network setup than how the controller responds to new flows, and we probably intend to follow this up with another type of attack.

3.5.3.4 Determining whether a Flow Rule Will Be Created with Wildcards (aggregation) or as an Exact Match

The objective of this attack (once again against the data flow *Asynchronous Message*) is more subtle: Rather than simply determining whether a flow rule will be created or not, instead determine the granularity of the created flow rule. It may be either matched to a single flow, resulting in a separate flow rule for every flow through the switch, or a subset thereof, which is a *reactive strategy*. Alternatively, flow rules may be created to cover several rules at a time, a *proactive strategy*. Realistically, some combination of the above may be used; it is also possible that an *adaptive strategy* is in use. Such a strategy would change between reactive and proactive "extremes" depending on the circumstances. In particular, it may decide that a number of similar flow rules may be *aggregated* into a single flow rule.

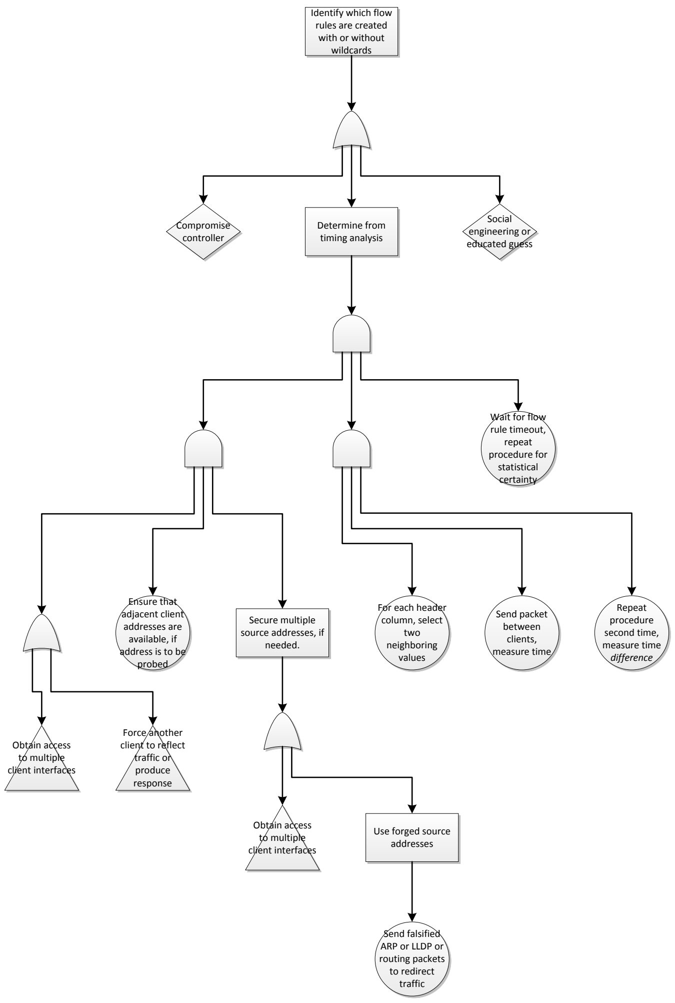

Identify which flow rules are created with or without wildcards
Compromise controller
Determine from timing analysis
Social engineering or educated guess
Wait for flow rule timeout, repeat procedure for statistical certainty
Ensure that adjacent client addresses are available, if address is to be probed
Secure multiple source addresses, if needed.
For each header column, select two neighboring values
Send packet between clients, measure time
Repeat procedure second time, measure time *difference*
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Obtain access to multiple client interfaces
Use forged source addresses
Send falsified ARP or LLDP or routing packets to redirect traffic

Figure 3.7: Subtree for determining whether aggregation is in use

3.5.4 Denial of Service

Attacks here will be performed by generating a very large number of packets, possibly instantiating new flows. Considering the switch, possible targets for attack are the *Flow Table*, which we may attempt to fill, and the control plane (*OpenFlow Module*, *Secure Channel* or *OpenFlow Interface*). Based on a reference implementation[41], it is safe to assume that part of the switching operation will be carried out in hardware, and part in software. Depending on how the switch is implemented, this may present a possibility to overload the CPU of the switch with operations that are implemented in software, including the sending of packets to the controller. It is this last operation that also provides avenues of attack against the controller as well. Overloading such a controller may be far more critical, if it operates multiple switches. Unless the controller has a security fault that we may exploit, we are forced to attempt to overload its network connection and/or processing capabilities. A critical shortcoming of the OpenFlow 1.0 specification is its reaction to the overflow of an interface input buffer, which we may attempt to put into effect by creating a very large number of flows simultaneously. The reaction of the switch is, according to the specification, to send entire packets rather than headers[24], which will result in us consuming the same amount of bandwidth on the management connection as on the data plane. Even if we are unable to overflow an input buffer, the sum of several attacks on various interfaces of the switch may be enough to result in an overload of the switch's management interface. Either

[24]Presumably to prevent packets from being lost.

attack will result in the inability to establish new flows. If we can obtain direct access to the management interface, a variety of other attack types maybe attempted. We may try to break the TCP connection with an RST attack (by predicting the TCP sequence number), or attempt a SYN flood.

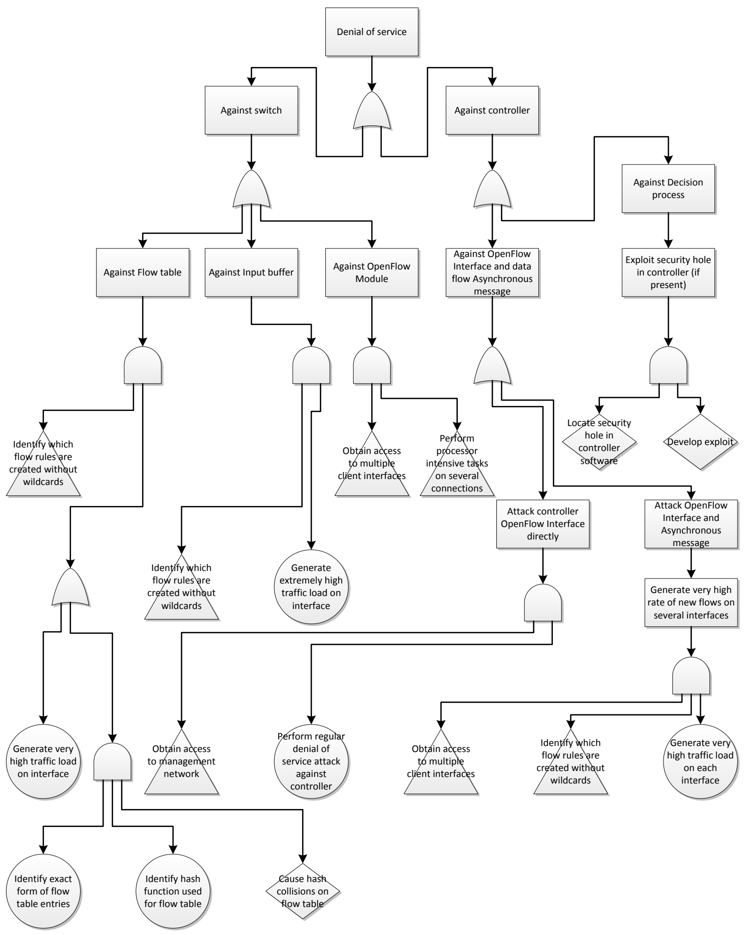

Denial of service
Against switch
Against controller
Against Decision process
Against Flow table
Against Input buffer
Against OpenFlow Module
Against OpenFlow Interface and data flow Asynchronous message
Exploit security hole in controller (if present)
Identify which flow rules are created without wildcards
Obtain access to multiple client interfaces
Perform processor intensive tasks on several connections
Locate security hole in controller software
Develop exploit
Attack controller OpenFlow Interface directly
Attack OpenFlow Interface and Asynchronous message
Identify which flow rules are created without wildcards
Generate extremely high traffic load on interface
Generate very high rate of new flows on several interfaces
Generate very high traffic load on interface
Obtain access to management network
Perform regular denial of service attack against controller
Obtain access to multiple client interfaces
Identify which flow rules are created without wildcards
Generate very high traffic load on each interface
Identify exact form of flow table entries
Identify hash function used for flow table
Cause hash collisions on flow table

Figure 3.8: Denial of service attack tree

3.5.4.1 Flow Table

In order to attack the *Flow Table* of the switch, it is necessary to have the controller generate a large number of new flow rules during a short time. Each new flow rule has a timeout, after which it will be removed. Although there are soft and hard timeouts, in practice only the soft timeouts are relevant, as we will not send repeat packets to keep a flow alive. We assume that each new packet results in the instantiation of a new flow. We must take into account the strategy of the switch: Is it *proactive* or *reactive*? A *proactive* strategy creates flow rules that encompass a broad category of flows, while a *reactive* strategy creates flow rules for each distinct flow. A *reactive* strategy allows the flow table to be overflowed more readily and will be modelled here. It has the advantage of finely grained access control, while having the disadvantage of creating a greater load on the controller. It also results in a trade-off for the flow rule timeout: A long timeout, lending itself to a *proactive* strategy, will allow the flow table to be overflowed more readily; a short timeout, lending itself to a *reactive* strategy, will on the other hand generate a higher load on the controller, which may open up other avenues of attack. For a steady state (i.e. same number of flow rules being created per time unit as being dropped) we have:

$$R_{flow\,rule} = \begin{cases} \frac{R_{clnt}}{s_{pkt}} & \frac{R_{clnt}}{s_{pkt}} < \frac{R_{mgmt}}{s_{hdr}} \\ \frac{R_{mgmt}}{s_{hdr}} & \frac{R_{clnt}}{s_{pkt}} \geq \frac{R_{mgmt}}{s_{hdr}} \end{cases} \tag{3.9}$$

$$R_{flow\,rule} > \frac{n_{flow\,table}}{t_{flow\,table}} \tag{3.10}$$

where

$R_{flow\,rule}$	is the maximum number of flow rules that can be generated per second
R_{clnt}	is the throughput available between the client system performing the attack and the switch
R_{mgmt}	is the throughput available between the switch and the controller, including overhead for encryption
s_{pkt}	is the size of packet required to effect the installation of a new flow rule
s_{hdr}	is the size of the encapsulated header sent to the controller
$n_{flow\,table}$	is the maximum number of entries in the flow table and
$t_{flow\,table}$	is the average dwelling time of a flow rule in the flow table (i.e. the flow rule timeout).

If this criterion is met, then an overflow of the flow table will occur. Note that the second case, i.e. $\frac{R_{clnt}}{s_{pkt}} \geq \frac{R_{mgmt}}{s_{hdr}}$ will eventually also result in the input buffer overflowing, leading us to an attack on the input buffer instead.
Depending on the implementation of the flow table in the hardware and/or software of the switch, other avenues of attack may exist. In general, a flow table will be implemented with the aid of a hash table[25]. A hash table in general provides lookups with amortised $O(1)$ performance, compared to a self-balancing binary search tree which offers $O(log\,n)$ performance, but this is guaranteed even for worst case input data. Any hash table is vulnerable to hash collisions; these are generally resolved using a linked list. Linked lists have an $O(n)$ lookup and insertion time, but as the hash function is assumed to be uniformly distributed and the hash table size is proportional to the maximum flow table size, this results in an essentially constant number of lookups for each query in the average case. However, if it is possible to cause hash collisions to occur predictably, the hash table will degenerate into a linked list, demonstrating worst case performance, i.e. $O(n)$ lookup time. For a large value of n, this may degrade performance considerably, especially considering the requirement that the lookups occur at line rate. [8] gives further information about this sort of attack, which may also be relevant to the controller, in case it stores system state in a hash table.

3.5.4.2 Input Buffer

Rather than attempting to overflow the flow table, we may attempt to overflow the *Input Buffer* (of a network interface) instead. Due to the semantics of the OpenFlow protocol, if the switch is fully compliant then whole packets will be sent to the controller rather than just headers, most likely to avoid data loss if no buffer is available. This will make a denial of service attack against the controller much more straightforward. Assuming that the switch does not drop any packets by itself, but always forwards them to the controller, we have:

$$\frac{R_{clnt}}{s_{pkt}} \geq \frac{R_{mgmt}}{s_{hdr}} \tag{3.11}$$

as the criterion for an overflow of the input buffer and

$$\frac{s_{ibuf}}{s_{pkt}} < n_{flow\,table} \tag{3.12}$$

with

[25]This is the case in Open vSwitch, which is the basis of Mininet. This can be verified by examining the source code.

s_{ibuf} as the input buffer size

being the differentiating criterion between an overflow of the flow table or of the input buffer. In reality, the switch is likely to drop packets, either at random or due to some criteria (such as QoS). Depending on which interface the attack is being performed on, this may or may not prevent the objective of the attack from being attained. If the client system is attached directly to the switch, only its own traffic will be affected by this behaviour. If the interface is connected to a number of other systems, then all of these will be unable to communicate via the switch. In the most extreme (but realistic) case, where the affected interface is connected to the rest of the Internet, Internet connectivity may be lost. In such a case, a protective mechanism which drops packets according to some predictable criteria, or at random, will help to accomplish the goal of the attack.

3.5.4.3 OpenFlow Module

It also possible that the software component (i.e. *OpenFlow Module*) may be attacked. As noted in 3.5.3.2, operations which are performed in software take longer. Unlike the data path, which is essentially “guaranteed” to run at line rate, the software component of the switch, insofar as it is used for forwarding operations, may be a bottleneck. It is necessary to know about the switch internals in order to perform such an attack, and considerably more traffic may be required than other denial of service attack possibilities. However, the attack does not depend on having flow rules installed, and therefore does not depend on a particular policy being followed.

3.5.4.4 Management Interface and/or Controller

Instead of considering the switch as a target, we may also consider the switch-controller interface, i.e. the *Secure Channel* or *OpenFlow Interface* elements. In the case of a single switch and controller setup, as we have modelled here, there is little difference in effect. But if, as seems likely in a production system, the controller is responsible for multiple switches, attacking the controller would have effects on all of them. This would make it a high value target for any attacker wishing to cause maximum disruption. Any such attack would use the same methodology as an attack on the flow table or input buffer. An attack that manages to overflow the input buffer of the switch would be particularly effective (as noted in the previous section), as it would amplify the effective traffic level sent to the controller. The possibility of attacking the controller through multiple interfaces, or even multiple switches, must also be taken into consideration. For multiple interfaces $1 \dots k$, we have

$$\sum_{i=1}^{k} \frac{R_{clnt,i}}{s_{pkt,i}} \geq \frac{R_{mgmt}}{s_{hdr}} \tag{3.13}$$

which allows us to overwhelm the switch-controller connection more easily, especially considering the extra overhead (encapsulation, encryption) that this connection requires. In the case of multiple switches, the effect of denial of service attacks against all of them simultaneously would be cumulative, leading to a very high load on the controller. In this case, even if the management interface could handle the traffic, the controller software itself may not be able to.

3.5.5 Attack Prerequisites

Some attacks have certain prerequisites in order to be executed successfully. These are discussed below.

3.5.5.1 Access to Multiple Client Interfaces

It may be necessary to have access to two or more clients, or at least be able to communicate with two clients across the switch. This is required for timing analysis; it is also beneficial to have multiple ports available for a denial of service attack. There are fundamentally two approaches: Firstly, we could attempt to gain control of a client system connected to the switch. Alternatively, we could try to get access to the network itself.

For the first approach, possibilities include social engineering, or an exploit of some software running on the system. We may also choose to get help from an administrator who has access to the system. The execution of such attacks is beyond the scope of this thesis.
For the second approach, we may attempt to gain physical network access - which means actually attaching a client system or access point, if the network is wired. This is high risk, and reasonable physical security should make this difficult[26]. Any hardware attached without authorisation will also attract the attention of any security software in use. If there is wireless access available, this may prove an alternative, especially if it is public (unencrypted), which is not uncommon on enterprise networks. A public WLAN is probably separated from the internal network by a firewall, but this does not concern us if traffic is being transmitted through the switch in question. Another possibility is the acquisition of access through virtual means, if VLANs are in use and we can somehow get network traffic to a switch which processes VLAN tags. A spoofed tag would then allow us access to the internal network.

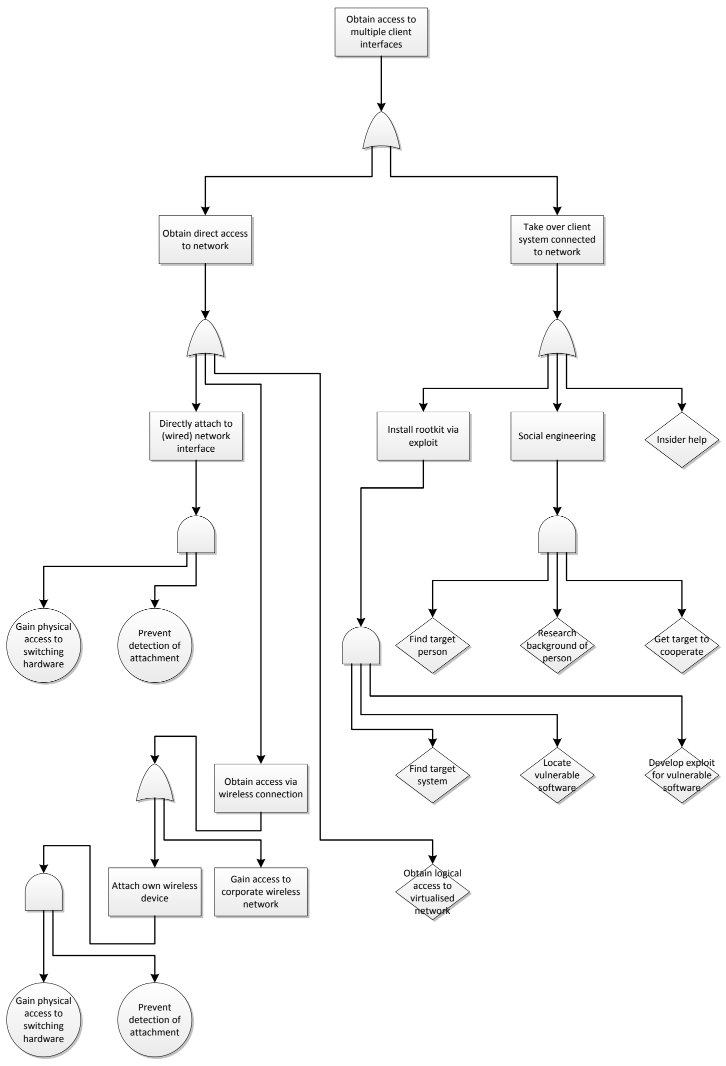

Obtain access to multiple client interfaces
Obtain direct access to network
Take over client system connected to network
Directly attach to (wired) network interface
Install rootkit via exploit
Social engineering
Insider help
Gain physical access to switching hardware
Prevent detection of attachment
Find target person
Research background of person
Get target to cooperate
Find target system
Locate vulnerable software
Develop exploit for vulnerable software
Obtain access via wireless connection
Attach own wireless device
Gain access to corporate wireless network
Obtain logical access to virtualised network
Gain physical access to switching hardware
Prevent detection of attachment

Figure 3.9: Subtree for multiple client interface access

3.5.5.2 Access to Management Interface

If access to the management interface is available, a number of attacks become feasible that would otherwise not be available to an attacker. A SYN flood, TCP reset or TCP connection hijacking attack could disable the connection between the switch and the controller, for potentially much less effort than other forms of attack. If we can detect when information is being transmitted -

[26]Also, much more damaging attacks may be possible if physical access is possible

even if we cannot decrypt it - then we can more accurately probe the response of the controller to packets.

It is safe to assume that information transmitted over the management interface is secure. We assume that the switch is located in a secure environment, so physical access is very difficult. If the management network is implemented as a virtual network - which is realistic in a distributed setting - it may be possible to get logical access, if we can get access to trunk networks where VLAN tagging is in use. Depending on what is connected to the management network, it may be possible to compromise a system that is being used for administration purposes, or some other system that is connected to the management network, which may be used for other network appliances. Risks arising from the administrator him- or herself[27] are generally not taken into consideration here. They may still be of interest to an attacker, and must therefore be considered by a defender.

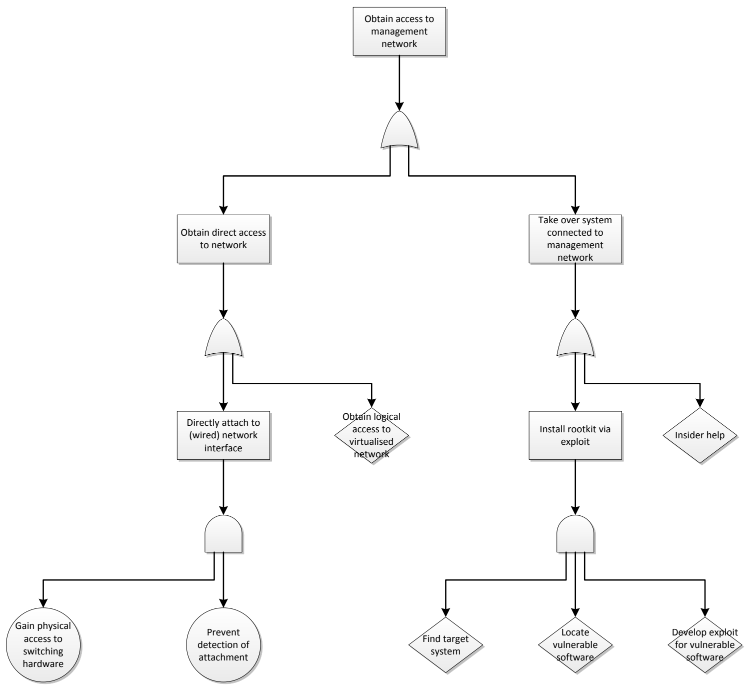

Obtain access to management network
Obtain direct access to network
Take over system connected to management network
Directly attach to (wired) network interface
Obtain logical access to virtualised network
Install rootkit via exploit
Insider help
Gain physical access to switching hardware
Prevent detection of attachment
Find target system
Locate vulnerable software
Develop exploit for vulnerable software

Figure 3.10: Subtree for management interface access

3.5.5.3 Effecting a Predictable Response from a Target System

If it is not possible to take over a client system, a timing analysis attack may still be possible if it possible to “force” the client to produce a predictable and consistent response. Essentially, we must choose a service running on the target system that will produce a consistent response. We should know the size of the response, or at least be able to predict it. This allows us to determine the expected transmission time. For small packets, where the propagation time is long compared to the transmission time, we may disregard this. UDP can be used for this purpose. TCP requires a handshake before data can be transmitted - here, the handshake itself will be small and therefore usable for timing analysis. ICMP could conceivably be used, but the controller may treat it differently from other traffic[28], or the host may choose to ignore it, resulting in unsatisfactory results.

[27]Such as vulnerability to social engineering or intentional collaboration with an attacker
[28]A controller may not install a flow rule in response to an echo request, and use an *ofp_packet_out* message instead, or block it altogether.

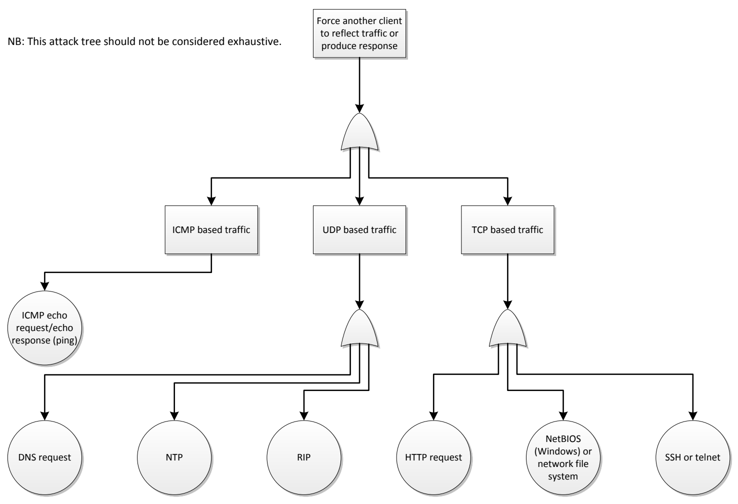

NB: This attack tree should not be considered exhaustive.
Force another client to reflect traffic or produce response
ICMP based traffic
UDP based traffic
TCP based traffic
ICMP echo request/echo response (ping)
DNS request
NTP
RIP
HTTP request
NetBIOS (Windows) or network file system
SSH or telnet

Figure 3.11: Subtree for producing a predictable response (for timing analysis)

Chapter 4

Changes Introduced in Newer Versions of OpenFlow

4.1 Introduction

In this chapter, the changes made to OpenFlow since the first production specification (1.0, as analysed in Chapter 3) was released are examined for potential security implications. In Section 4.2, a short history of OpenFlow specifications is presented. The subsequent sections evaluate the security implications of the changes introduced in each of the production specifications.

4.2 History

Presently, OpenFlow is managed by the *Open Networking Foundation*[46], a consortium of IT companies and network hardware manufacturers. After a period of development, the first release of the OpenFlow specification was made at the end of 2009; to-date this remains the most widely implemented one, especially in hardware. There is also a set of specifications for the *OpenFlow Configuration and Management Protocol*, but this is only used to configure switches and has little relevance, assuming that switches are managed from a separate network. Table 4.1 shows the different versions of the OpenFlow Switch Specification.

Specification	Published	Wire Protocol	For production use
OpenFlow Switch Specification 1.3.1	06.09.2012	`0x04`	✓
OpenFlow Switch Specification 1.3.0	13.04.2012	`0x04`	✓
OpenFlow Switch Specification 1.2	05.12.2011	`0x03`	✓
OpenFlow Switch Specification 1.1.0	28.02.2011	`0x02`	✓
OpenFlow Switch Specification 1.0.0	31.12.2009	`0x01`	✓
OpenFlow Switch Specification 0.9.0	20.07.2009	`0x98`	✗
OpenFlow Switch Specification 0.8.9	02.12.2008	`0x97`	✗
OpenFlow Switch Specification 0.8.2	17.10.2008	`0x85`	✗
OpenFlow Switch Specification 0.8.1	20.05.2008	`0x83`	✗
OpenFlow Switch Specification 0.8.0	05.05.2008	`0x83`	✗
OpenFlow Switch Specification 0.2.1	28.03.2008	`(1)`	✗
OpenFlow Switch Specification 0.2.0	28.03.2008	`(1)`	✗

Table 4.1: Versions of the OpenFlow Switch Specification

Each specification has a *wire protocol*, which indicates the version of the protocol syntax; specifications sharing a wire protocol version have the same syntax, although the *semantics* may have changed. The OpenFlow protocol is backwards compatible in the sense that any controller and any switch that follow the specification are capable of interacting - they will utilise the latest protocol version supported by both. At any rate, a list of supported features is exchanged as part of the protocol handshake (*OFPT_FEATURES_REQUEST*), allowing implementations to negotiate the use of features beyond those required by the specification.

4.3 OpenFlow Switch Specification 1.1.0

4.3.1 Outline of Changes

- Support for multiple flow tables
- Support for group table
- Support for multiple layers of VLAN tags
- Support for virtual ports (collections of physical ports)
- Removal of support for emergency flow cache
- Support TTL decrementation
- Support for SCTP, ECN rewriting
- Message handling semantics defined (out of order handling)

In the following subsections, we highlight the most important changes in v1.1.0, and whether they are security-relevant or not.

4.3.2 Multiple Flow Tables

The 1.0 specification does not support multiple tables, at least not from the perspective of the controller - the switch may use multiple tables internally. The addition of support for multiple flow tables was intended to allow the OpenFlow controller to better utilise the hardware capabilities of the controller, as well as allowing different layers of flow rules. These could be used for different purposes, for instance access control or traffic prioritisation. This would tend to reduce the number and complexity of flow rules. The 1.1 specification requires that metadata and state information be stored for each packet as it is processed by the various flow tables, which have an index starting at zero, allowing subsequent flow tables to extend or modify the action sets based on previous tables' flow rules. Also, flow rules can specify that packets be forwarded to another flow table. It is only permitted to forward to tables with a higher index, preventing the existence of loops. After each table, an action can be added to the *action set.* When there are no more tables to forward to, or a flow rule does not specify which is the next table to be used, the accumulated action set is executed. It is also possible to execute actions immediately, without affecting the action set. These changes increase complexity (and therefore processing time, especially under load). It is therefore probably advisable to make use of a formal validation if complicated, multi-table setups are to be used. In particular, it should be shown that accumulated actions from multiple tables are not contradictory.

4.3.3 Group Table

The 1.1 specification adds support for a *group table.* The group table contains groups, each having a list of actions, forming an *action bucket.* Groups and buckets both have counters attached. There are four types of groups: *All*, *select*, *indirect* and *fast fail-over.* The *all* type group executes all of the action buckets. It is intended to implement multi- and broadcast forwarding. The *select* type group executes one action bucket, the selection of which is implementation dependent. It is intended to allow forwarding along multiple paths for redundancy purposes. The *indirect* type is the same as an *all* type with just one bucket. It is intended to be used to simplify setups where multiple flow rules perform the same (complicated) action. The *fast failover* type forwards to the first action bucket marked as live, the liveness attribute being associated with a port or group of ports. As the name suggests, it can be used to perform alternate routing without first contacting the switch. These mechanisms can be used to increase the resilience of the OpenFlow system with less complexity.

4.3.4 Emergency Flow Cache

The 1.0 specification (section 4.3) requires that the flow table have a field for an *emergency bit.* In case the switch loses contact with the controller[1], all flow table entries are deleted apart from those marked with this bit. These rules must be manually removed by the controller if no longer desired. The 1.1.0 specification removes this feature and instead adds two failure modes: *Secure* and *standalone.* In the secure mode, the switch behaves as if *all* of the flows were marked with the emergency bit. The switch keeps its set of flow rules, the only difference being that packets are no longer forwarded to the controller. Flow rules maintain their timeouts, and so may eventually expire. The standalone mode makes the switch behave as an ordinary Ethernet switch, requiring that this functionality be available in the switch software or firmware.
These changes are security relevant: Any denial of service attack or transient failure which results in a loss of connectivity to the controller will result in the switch behaving differently. The standalone mode ensures continued connectivity, but may result in security properties becoming compromised, allowing unauthorised access to network hosts. On the other hand, the secure mode may result in a gradual loss of availability. The new behaviour does have the advantage of greater simplicity.

4.3.5 Message Handling Semantics

The OpenFlow 1.1 specification introduces the requirement that the control channel be *reliable*, in that delivery of control messages is guaranteed. This does not represent any change in practice, as the control channel is generally implemented over TCP[50], which itself guarantees delivery, although it does clarify that in the case of switch failure, *no* assumption can be made about switch behaviour. All messages from the controller to the switch must be processed, and an error message must be sent to the controller if processing is not possible, though in the case of an *ofp_packet_out* message the packet itself may be dropped silently under certain circumstances. The switch is required to report to the controller all internal state changes, including flow rule expiry. However, *ofp_packet_in* messages can be dropped in the case of congestion. The controller is completely unrestrained by the OpenFlow protocol; it is not required to respond to any messages sent by the switch. These changes may affect the behaviour the the switch and controller in the event of a denial of service attack.
The specification also notes that the switch may freely change the ordering of packets; the controller must *not* rely on the packets being processed in the input order. The barrier message can be used to ensure that messages are processed in a particular order, which is required especially if dependencies exist between messages. This possibility, as well as the potential reordering capability, introduce the aspect of *timing* into security considerations. As with all asynchronous systems, race conditions may be an issue, and this should be taken into consideration if controller policy results in interdependencies being introduced. As with 4.3.2, the use of formal validation is desirable.

4.3.6 VLAN Tags

The OpenFlow 1.1.0 specification allows frames containing multiple 802.1Q tags to be processed and the VLAN ID and QoS tag fields they contain to be matched against. Some network setups require multiple embedded layers of virtual networks and other network setups may be simplified by this functionality - it is possible to encapsulate multiple virtual networks inside another virtual network. Multiple tags may be managed as a stack, with push and pop actions supported. However, this makes a type of attack called *VLAN hopping* possible, where packets

[1]The switch would establish this due to lack of reply to an *OFPT_ECHO_REQUEST* message.

with multiple VLAN tags can be forwarded to other virtual network segments, if care is not taken when configuring the network[13].

4.3.7 Summary

The 1.1 specification represents the first revision after the original production specification. A number of the changes (e.g. 4.3.5) represent clarifications of behaviour which were previously not defined in the specification. The aforementioned modification merely made clear the generally asynchronous nature of the system, and the necessity for the explicit enforcement of ordering with barrier messages, if it is desired. Other changes (e.g. 4.3.4) are a result of the experience that certain features were not implemented in practice - the new behaviour is much simpler, also from a security perspective. The addition of support for TTL decrementation is possibly a correction of an oversight made during the original specification which prevented the use of OpenFlow for layer 3 switching. An issue concerning both this and future specifications is greater complexity: Multiple flow tables greatly increase the complexity of the processing pipeline, and introduce the possibility of inconsistency between different tables. The more complicated state and processing increases the potential for vulnerabilities. To a lesser extent the same principle applies to support for multiple VLAN tags or virtual ports. It is important that the added complexity is weighed against actual usage of the new functionality, so that additional complexity is not introduced merely to cover corner cases (which should be dealt with by extensions, or be turned off by default).

4.4 OpenFlow Switch Specification 1.2

4.4.1 Outline of Changes

- Extensible header field matching and rewriting
- Changes to *OFPT_FLOW_IN* syntax
- Extensible error messages
- Support for IPv6 (via new extensible field support)
- Changes to *OFPT_FLOW_MOD* semantics
- Facilitation of controller fail-over and load balancing
- Packet buffering can be turned off (*OFPCML_NO_BUFFER*)
- Max-rate queue property added

In the following subsections, we highlight the most important changes in v1.2, and whether they are security-relevant or not.

4.4.2 Field Matching and Rewriting

The OpenFlow 1.2 specification completely overhauls field matching and rewriting functionality. Instead of sending fields specified as part of a static header, the specification instead allows them to be specified in a data structure called *OXM* (OpenFlow Extensible Match). This structure stores the fields in a type-length-value format, allowing support for new fields to be added easily as well as allowing for arbitrary masks. This mechanism is also used to specify field rewrite actions. The type field has a class subfield, allowing for extensions to add new sets of fields; this could in particular be used to add support for matching application layer fields to OpenFlow (i.e. support for deep packet inspection), which would allow considerably more flexibility where OpenFlow is being used to enforce access control or ease lawful interception of the packet streams.

4.4.3 Packet Buffering

The OpenFlow 1.2 specifications allows the controller to deactivate packet buffering on the switch using *OFPCML_NO_BUFFER*. The switch is then required to send full packets to the controller, imitating the existing behaviour in the event of the buffer being full. The deactivation of buffering on the switch may considerably increase the load on the controller-to-switch connection, although it trivially precludes the possibility of the input buffer overflowing. The utilisation of this functionality might allow an attacker to overload the secure channel or the controller with large packets.

4.4.4 Multiple Controllers

The OpenFlow 1.2 specifications establishes support for multiple controllers for a single switch, allowing load balancing and fail over. Virtualisation does not require these mechanisms and can be supported with the 1.0 specification. Controllers are now assigned one of three roles: *Master*, *slave* or *equal*, the last being the default option. A switch can connect to any number of equal or slave controllers, but only one master controller is supported. The switch maintains connections to all controllers, accepting controller-to-switch messages from each as permitted by the controller's role, and sending asynchronous messages to all relevant controllers. The switch does not perform arbitration, therefore hand-over to other controllers needs to be implemented on the controller. Moreover, the controller may change its role at any time. A controller in the master or equal role may send any message to the switch and will receive any relevant response generated by messages it sent. A controller which becomes a master will relegate the previous master to slave role, which entails read-only access to the switch; any operation which changes the switch state is prohibited.

Support for multiple controllers increases the robustness of the OpenFlow system considerably: The lack of a single point of failure makes denial of service attacks less effective, and the use of load balancing increases scalability, also allowing more effective handling of large loads. Questions arising from a security perspective include the possibility of race conditions, if controllers do not synchronise their actions properly, as well as questions on the impact of timing analysis attacks; it may be possible to determine which controller is controlling a given switch if the network topology is asymmetrical (a form of information disclosure attack). The security impact could only reasonably be assessed when the setup and topology of the controllers is defined.

4.4.5 IPv6 Support

The OpenFlow 1.2 specification introduces support for matching and rewriting IPv6 as well ICMPv6 header fields. This capability is added via the flexible matching described in 4.4.2. Support for IPv6 is a logical and necessary extension to the OpenFlow, if it is to be used after the transition to IPv6. IPv6 introduces new mechanisms for address resolution (c.f. neighbourhood discovery[42]) which should be examined for spoofing possibilities. These may lead to tampering attacks, as discussed in 3.5.2.2.

4.4.6 Summary

A major theme in this revision is the increase in flexibility: We have 4.4.2 and 4.4.4 as the primary changes. The flexible matching support also enables support for IPv6 (4.4.5). The matching may have security relevance, especially if payload matching is used in flow rules. The support for multiple controllers is a major novelty which offers the possibility of redundancy and some limited forms of load balancing, although the OpenFlow Switch Specification 1.3.0 increases this capability considerably. Although this may be a generally positive contribution to the security of OpenFlow, the use of multiple controllers, especially the critical transition phase when control is transferred from one to another, must be examined in order to discover any possible race conditions.

4.5 OpenFlow Switch Specification 1.3.0

4.5.1 Outline of Changes

- Redesign of capabilities negotiation during handshake
- More flexible handling of unmatched packets (special flow table entry)
- IPv6 extension header handling (detection of presence in packet)
- Flow rules can now specify rate limiting
- Controller can now specify filters for asynchronous messages
- Allow auxiliary connections to controller in order to enable parallelism
- Statistics include duration field, counters can now be disabled

In the following subsections, we highlight the most important changes in v1.3.0, and whether they are security-relevant or not.

4.5.2 Unmatched Packets

The OpenFlow 1.3.0 specification changes the reaction of the switch in case of a packet not matching any flow rule (in a given table, in case multiple tables are in use): Instead of having flags to specify behaviour in this case, a new fallback default flow rule is used instead. This rule's actions are then executed. This change allows more flexibility when dealing with unmatched packets, since it is for example possible to have these processed by a switch's conventional pipeline, if the switch is a hybrid rather than a pure OpenFlow switch. In the absence of any default flow rule, the packet is simply dropped. This may be of use in case the switch is under severe load, although this behaviour may also allow information to be derived about the contents of the flow table and/or switch state.

4.5.3 IPv6 Extension Header Handling

The OpenFlow 1.3.0 specification adds (via OXM, see 4.4.2) support for matching the presence of the various IPv6 header extensions. This change should allow more flexibility when enforcing policy in IPv6 networks. For instance, it would be possible to reject packets containing a routing header.

4.5.4 Meters

The OpenFlow 1.3.0 specification adds a new data structure called a *meter table*. Each entry in the table is a *meter*, each meter having one or more *meter bands*. The meters are used by applying an action to an incoming packet. Any number of flow rules can use one meter, and all packets sent to a meter will result in the meter being updated accordingly. Once a meter band is exceeded, the band is triggered and the action is performed on all packets sent to the meter, until the next band is triggered. The action would most likely be to drop the packet, although it can also be used to implement DiffServ and similar QoS strategies. This would also apply to the fallback rules introduced in 4.3.4. The meters also have counters; these are *not* used for enforcing the bands but rather for informational purposes. A major application of the feature is to limit the number of messages sent to the controller, which would be invaluable in preventing denial of service attacks. This feature could also allow countermeasures against denial of service attacks targeting client systems in the OpenFlow network.

4.5.5 Event Filtering

The OpenFlow 1.3.0 specification allows all controllers to add a filter to the switch-controller connection using an *asynchronous configuration* message. This prevents the switch from sending certain asynchronous messages to the affected controller - messages which can be filtered are *packet in*, *port status* and *flow removed*. The controller can specify which of these message types it does not want to receive; it is also possible to install separate filters for different controller roles (master and equal versus slave). This functionality can improve scalability, allowing separate controllers to handle different event types, as well as decreasing the load on the individual controller caused by asynchronous messages.

4.5.6 Auxiliary Connections

The OpenFlow 1.3.0 specification introduces the possibility of having multiple, parallel connections between the switch and the controller. One of the connections will be designated as the main connection (the one with a connection ID of zero), all of the others are considered auxiliary connections and may be used only when the main connection is active (i.e. they are not “backup” connections). Auxiliary connections must share the same source IP as the main connection, but the port and transport mode may be different. Certain transport modes do not guarantee reliable delivery of messages (e.g. UDP), and the OpenFlow protocol does not itself provide any method for ensuring delivery. Alternative transport modes may also deviate from in-order delivery and there is *no* defined ordering between two channels; in order to ensure that two messages are delivered in a given order, a reliable transport protocol must be used (e.g. plain TCP or TLS), the messages must be sent over the same channel *and* a barrier message must be placed between them. The intent of this feature is to improve performance and exploit parallel designs in switches by allowing packet-in and packet-out messages to be sent concurrently[2]. Although the sending of concurrent messages introduces the possibility of race conditions, 4.3.5 already introduced this possibility, which must be taken into consideration by the controller. This change may allow the controller to cope with denial of service attacks more effectively by exploiting parallelism.

4.5.7 Summary

The OpenFlow 1.3.0 specification also increases flexibility and controllability, with the changes to handling of unmatched packets and the possibility of disabling counters (see 3.5.4.3). Support for matching IPv6 headers can be considered an extension of the support added in the previous specification. The new specification contains several new features which jointly address one of the major shortcomings of the OpenFlow specification, namely the issue of scalability. The support for flow meters has a potential to greatly reduce the impact of denial of service attacks, as well as merely incidental congestion. Likewise, the support for auxiliary connections allows greater performance as well as more parallelism, both on the switch and on the controller. Finally, support for controllers filtering events complements the multiple controller support added in the 1.2.0 specification, making it more useful for load balancing purposes. Unfortunately, the degree of control offered for the filter is quite limited.

4.6 OpenFlow Switch Specification 1.3.1

4.6.1 Outline of Changes

- Redesign version negotiation during handshake (send complete set of supported versions instead of just highest one)

In the following subsections, we highlight the most important changes in v1.3.1, and whether they are security-relevant or not.

4.6.2 Summary

There are practically no security-relevant changes in 1.3.1, which is to be expected given that it is only a minor release. It allows greater flexibility in finding the best common protocol version, although as the newer protocols are largely supersets of the older versions, and therefore it is not unreasonable to expect intermediate versions to be supported, this probably has limited applicability in practice.

[2]However, any message can be sent over the auxiliary channels

4.7 Conclusion

It is evident that the OpenFlow standard is growing in complexity - with flexible matching support, support for multiple flow tables and multiple controllers being the most important changes, in terms of overall impact. This increases attack surface, making it more difficult to assure security properties. On the other hand, some of the changes also mitigate issues discussed in this thesis (see Chapter 6 for more information) and it is clear that the security implications of the use of OpenFlow in production networks are becoming apparent to the community. It is to be expected and hoped that future specifications take security into account to a still greater extent.

Chapter 5

Experimental Examination

5.1 Introduction

In this chapter, working exploits for several of the security issues discussed in Chapter 3 will be developed. This is in order to demonstrate the practical applicability of the issues discussed here as well as to gain insights into possible mitigations and remedies for them. Furthermore, the creation of a virtual network simulation also demonstrates the use of OpenFlow in practice, and permits other potential problems pertaining to deployment to be discovered.
This chapter is further subdivided into the following sections: Section 5.2 examines the different issues regarding exploitability, Section 5.3 describes the simulation setup and Section 5.4 describes the execution and presents the results of the attempted exploitation.

5.2 Evaluation of Security Vulnerabilities

In order to choose a vulnerability to exploit, it is necessary to define certain criteria. Firstly, the vulnerability should be *practical* to exploit, that is, with the available tools it should be possible to perform the operation in a reasonable period of time. Secondly, the vulnerability should be *reliably* exploitable, that is, it should be possible to perform the exploit consistently, without relying too much on factors that we have no control over. Thirdly, the exploit should be *novel*, that is, it should be possible, or feasible, only with OpenFlow, and should not be trivial. Finally, the attack should have notable *consequences*. It is therefore necessary to analyse the vulnerabilities discussed in Section 3.5.

5.2.1 Tampering

In 3.5.2 (*Tampering*), there are essentially two classes of vulnerabilities described: 3.5.2.1 (*Counters*) describes attacks based on overflowing counters. As demonstrated with Equation (3.1), this attack is practically impossible to carry out, given the range of the counters and the available bandwidth. The effects of such an attack, even if performed successfully, are also too limited to make this attack an attractive target. 3.5.2.2 (*Redirecting Flows to Another Port*) and, to a lesser extent, 3.5.2.3 (*Having Packets Modified*), describe attacks based on the aggregation of flow rules and reactive specialisation. These attacks are novel - they are only possible in software defined networking scenarios - and also have a high impact, potentially allowing man-in-the-middle attacks to be performed. However, they rely on the controller software and the strategy that it uses: A purely reactive strategy would not exhibit the vulnerability. Also, there are certain demands on the network setup - at least two systems connected to different ports are required in order to perform the attack in a meaningful way.

5.2.2 Information Disclosure

3.5.3 (*Information Disclosure*) describes several attacks in detail, all of which are based on the concept of timing analysis. These attacks are novel, not being possible on conventional networks. The impact of information disclosure is moderate in a typical scenario, although such information may be of use in performing other attack types, not only those related to OpenFlow. The attack described in 3.5.3.2 (*Determining what Action a Flow Rule Will Take*) is hardware-dependent and would require a very large number of samples to carry out in practice. The attack 3.5.3.1 (*Determining whether a Flow Rule Exists*) is relatively straightforward to carry out, although one may argue that the impact is small, unless the flow rules cover a large number of flows. 3.5.3.3 (*Determining whether a Flow Rule will be Created*) is also straightforward, but does not provide much information, as it is obvious to the observer if the controller does not permit that transmission of a packet. 3.5.3.4 (*Determining whether a Flow Rule Will Be Created with Wildcards (aggregation) or as an Exact Match*) provides useful information for other attacks described here, although the execution is more complicated.

5.2.3 Denial of Service

3.5.4 (*Denial of Service*) lists attacks which are all based on generating a large amount of traffic towards the switch in order to overload some aspect of the OpenFlow system. These attacks are only moderately novel - denial of service attacks are possible on any network. The impact depends on the exact network setup, but if an OpenFlow switch (or a group of switches) is used for a backbone or other high load network, the impact could be substantial. 3.5.4.1 (*Flow Table*) and 3.5.4.4 (*Management Interface and/or Controller*) describe essentially the same attack, albeit with different consequences. It may be of interest to test this difference in practice. Equation (3.10) gives a criterion to distinguish between 3.5.4.1 (overflow of *Flow Table*) and 3.5.4.2 (overflow of *Input Buffer*). 3.5.4.2 (*Input Buffer*) may be of particular interest due to the specified behaviour of OpenFlow in the event of the input buffer overflowing - entire packets are sent rather than just packet headers - and this may have an additional amplifying effect on any attack. 3.5.4.3 (*OpenFlow Module*) is a special case, which is dependent on the switch performing certain operations in hardware, and this aspect of its operation constituting a significant bottleneck. It does have the advantage of not being able to be prevented by the use of some form of rate limiting, which would be of assistance in the case of 3.5.4.1 (*Flow Table*) and 3.5.4.4 (*Management Interface and/or Controller*)[1]. 3.5.4.4 (*Management Interface and/or Controller*) is essentially the same attack as 3.5.4.1 (*Flow Table*) and 3.5.4.2 (*Input Buffer*), but with the effects observed at the controller. In our setup, we will only use one switch, but in general, any attack with detrimental effects on the controller rather than the switch alone would have a considerably higher severity. Also described in the attack tree, but not in the text, was the possibility of exploiting any security issues in the controller. We should not count on locating any such issues, and even if we are able to, this type of attack lacks general applicability.

5.2.4 Conclusion

In order to proceed, it is necessary to decide which vulnerabilities are practically exploitable and which are not. There are no tampering issues which would be practical to exploit, given the timeframe and (especially) the requirement that no special software is needed. The information disclosure issues are more practical to exploit, but the impact is not necessarily very high, and multiple systems are required (or one system and the forced cooperation of another). The denial of service issues are the easiest to exploit, have a substantial impact, and are still somewhat novel to OpenFlow systems. Therefore, the preferred strategy is to attempt to exploit one of the denial of service vulnerabilities. This would also allow familiarity to be gained with the test setup before proceeding to exploits that may require more complicated designs. Later, we perform a more in-depth examination of information disclosure issues.

5.3 System Setup

5.3.1 Overview of Setup

The network setup chosen for test purposes reflects on one hand the setup described in Section 3.3, but also the need for simplicity. The setup consists of a number of identical client systems, a switch running OpenFlow software[2] and a separate controller. Each of these systems run in separate virtual environments. Each client and the controller has a unique virtual network

[1]Rate limiting is *not* a panacea, as it also affects innocent traffic

[2]As in the model, there is no fallback to conventional switching

connection to the switch. The attacker controls one or more client systems, depending on the attack in question. The attacker can make implicit use of other systems to perform attacks, but does not have any control over them, especially the switch and the controller. The attacker can only make observations through clients controlled by him or her. External observations (for instance, packet dumps between the switch and the controller) are not permitted for the attacker, but will be used in order to evaluate the impact of the attacks.

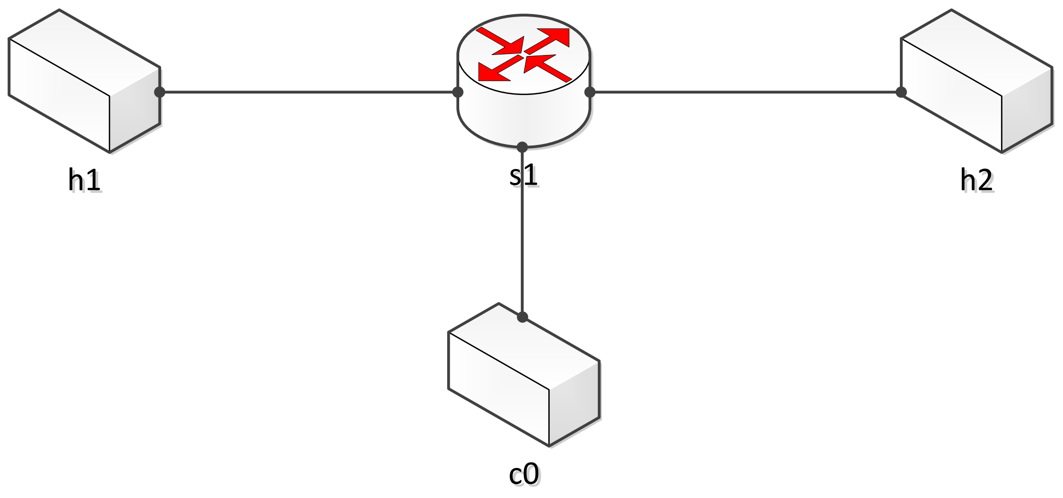


Figure 5.1: Schematic diagram of simple virtual network setup

Some forms of attack[3] require a more sophisticated network environment than the very simple one shown in Figure 5.1. For this reason, there is a second virtual network setup. In Figure 5.2, there are two virtual switches, which are linked together. Each of the switches has three further virtual hosts. A single controller controls both switches. As above, all of the data path links are identical in performance[4], while the control path links also have identical and distinctive (from the data path links) performance characteristics. This setup requires that the controller supports layer 3 forwarding properly.

5.3.2 Virtualisation Software

5.3.2.1 VirtualBox

In order to simplify the setup, as well as improve repeatability, a virtualised network setup will be used, as noted above. The underlying virtualisation solution chosen is *VirtualBox*[47]. On the hardware platform chosen, Ubuntu 12.04.1 LTS, VirtualBox makes use of the hardware virtualisation technology in newer processors (VT-x and AMD-V). The virtual machine is connected to the Internet over NAT (which is necessary to install and update software, amongst other things), while the *host-only adapter* functionality allows a secure connection to the host system, so that SSH can be used. The virtual machine has no window system installed, but X11 forwarding over SSH can be used to run GUI-based applications.

[3]That is, those involving information disclosure and/or tampering

[4]It is also conceivable that a “backbone” link between the switches be simulated, but for simplicity this possibility has not been taken into consideration

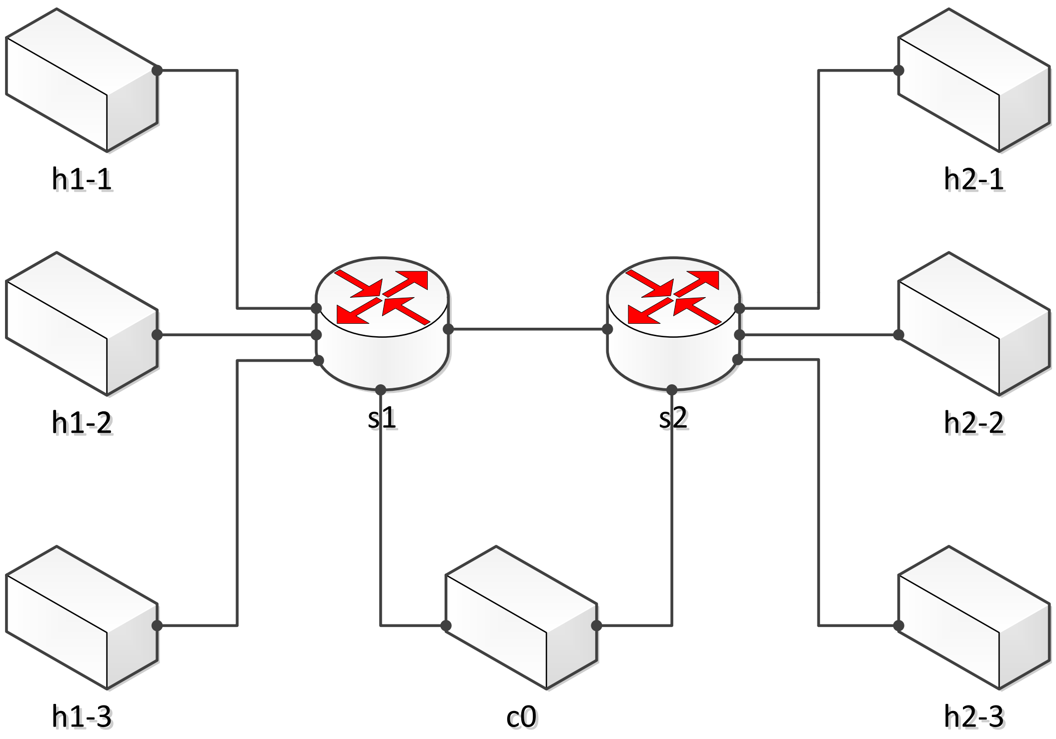


Figure 5.2: Schematic diagram of advanced virtual network setup

5.3.2.2 Mininet

The network virtualisation solution is *Mininet*[38], which is run on top of VirtualBox. Mininet is based on *network namespaces*, a feature of the Linux kernel. They can be thought of as a form of lightweight network virtualisation, where processes can only access resources inside their namespace[34], although the processes themselves do not exist in a separate memory space and also share the same filesystem. See [23] for more information on Mininet implementation.

Mininet is configured using the command line utility *mn*, which allows network topologies to be specified parametrically, by specifying hosts and switches. For each virtual network link, Mininet allows a variety of performance parameters to be specified; this is essential for simulating real world systems and a requirement to implement most of the attacks described in Section 3.5.

Mininet provides its own CLI, but for systematic use it is easier to use its Python API directly. The API supplies *topology* objects, which are used to construct *net* objects. Individual systems are modelled as *nodes* (which can be *hosts*, *switches* or *controllers*) and possess *interfaces*, representing network interfaces. A switch can be a kernel switch or a user switch (modelled by the *UserSwitch* class). A controller can be a NOX controller or a user-supplied controller (which must be started and stopped by the user), which is supported by the *RemoteController* class. A pair of interfaces share a *link*. Links can be created with parameters - or these can be be supplied to the topology, in which case, they will apply to *all* links - and these parameters include bandwidth, delay, buffer size as well as simulated packet loss (specified as percentage of total transmitted packets). This is implemented by the *TCLink* class. In addition to this, it is possible to create host nodes with performance constraints via the *CPULimitedHost* class. See Section D.3 and Section D.4 to see examples of how the Mininet API can be used.

As suggested by the name, the *TCLink* and *TCIntf* classes make use of the `tc` (for *traffic control*) command, which is supplied by the *iproute2* package, which also supplies the the `ip` command, an alternative to the well known `ifconfig` and `route` commands. If enabled, Mininet uses the *HTB* (Hierarchical Token Bucket[18]) queueing discipline to enforce bandwidth restrictions. Loss, delay and jitter are introduced with the *netem* (Network Emulator[65]) queueing discipline. Netem is also able to simulate other faults occurring in real networks, such as packet duplication and out of order arrival, but this functionality is not encapsulated by Mininet. The

Component	Parameters available
CPULimitedHost	Number of CPU cores
	Percent of processor time
TCLink	Bandwidth
	Delay
	Jitter
	Loss
	Speedup
	Maximum queue size

Table 5.1: Configurable performance parameters

CPULimitedHost class makes use of kernel *cgroups* (Control Groups[37]) to enforce performance limitations.

5.3.3 Measurements and Observations

It is possible to observe traffic in the system using standard tools such as *Wireshark*[66] or *tcpdump*[62], both of which are based on *libpcap*. Wireshark has explicit support for the OpenFlow protocol, allowing the contents of OpenFlow protocol packets to be analysed. All packets sent between the virtual switch and controller in the standard setup of Mininet are sent over a dedicated virtual interface. The use of filtering can make the OpenFlow packets easier to distinguish, and can also eliminate the *echo request* (type 2) and *echo reply* (type 3) messages, which are essentially just noise. The Mininet package includes the tools *ovs-ofctl* and *dpctl*, which can be used to dump the flow table, as well as add, remove and modify flow rules. It is also possible to obtain protocol statistics, including the number of different message types sent and received by the switch. The switch and controller also have logfiles; by default these are located at `/tmp/s1-ofd.log` and `/tmp/s1-ofp.log` (for a switch called *s1*) and `/tmp/c0.log` (for a controller called *c0*).

5.3.4 Attacking System

The attacking system is one of the client systems simulated by Mininet (and as such, the same VM that runs the controller and switch). It may be necessary to limit processor time for the attacking process, in order to avoid a situation where the process itself interferes with the operation of the switch or controller (the attacker will certainly not be running on the same system as the switch or controller - unless virtualisation is in use - and if it were, many other possibilities would exist to degrade performance).
To implement the attacker, the packet generation and analysis framework *scapy*[9] will be used. It allows the creation of packets with arbitrary data in the header fields. The Mininet system is set up by a script[5], which then executes the attacking script on one of the host nodes. A trace of the control traffic is performed using *tcpdump*. Afterwards, the system is shut down.
The facility *netcat*[21] can be used for a variety of network diagnostics as well as ad hoc network applications. In 5.4.2, it is used to simulate TCP clients and servers.

5.4 Execution and Results

5.4.1 Denial of Service

There are several potential scenarios which could be tested, of which 3.5.4.1 (*Flow Table*) and 3.5.4.2 (*Input Buffer*) are the most relevant and easiest to reproduce with the Mininet setup. The only difference between the two lies in the packet generation rates relative to available bandwidth, as illustrated in Equation (3.10). Essentially, to produce either scenario, it is necessary to generate

[5]See Section D.4

a large number of packets which will be sent to the controller and result in it installing a new flow rule for each packet. We use the default controller for the purposes of this attack; it implements a standard learning switch, exemplifying a purely reactive strategy. As the controller installs only rules matching header fields *exactly*, it is only necessary to permute some value in the header in order to effect the installation of a new flow rule. For this purpose, the source and destination port fields of a UDP packet are used. Section D.1 describes the attack script, which is executed by the framework in Section D.5.

Category	Direction	Description	Count
Protocol messages	Received	Hello	1
		Feature request	1
		Set configuration	1
		Flow removed	0
		Packet out	10271
		Flow modification	992257
	Sent	Hello	1
		Feature reply	1
		Packet in	994974
		Error	37517
Flow manipulation	Received	Add	992257
		Modify	0
		Delete	0
Error notification	Sent	Flow mod failed (all tables full)	37517

Table 5.2: Summary of `dpctl show-protostat` output on switch after test run on default controller

The attack script was executed with a bandwidth of 100 Mbps, a latency of 10 ms and no loss, and using one million packets. Examining the packet trace generated by *tcpdump*, it is clear that the flow table is eventually filled to capacity - in a scenario where the size of the flow table is more limited, or an attack is mounted by multiple clients simultaneously, this would be filled even more rapidly. No attempt to remove any flows is made by the controller. A number of *ofp_packet_out* messages are also recorded. By inspecting the contents of the controller logfile, it is evident that, after the controller has received a large number of flow modification failures (due to the flow table being full), it begins to ignore further *ofp_packet_in* messages. This appears to reduce the error rate, presumably resulting in the loss of the packets which were not forwarded. This does not prevent further errors from occurring completely. The controller logfile increased in size substantially during the attack. If no mechanism is in place to purge old logs on the controller, this could represent a denial of service attack itself by filling available disk space. This possibility was discussed in 3.4.1.5.
It is also possible to use alternative controllers - the POX controller is a modular design, based on Python[44]. For an initial test, POX was started with the *forwarding.l2_learning* module, which, as the name suggests, implements a simple learning switch. Using 640,000 packets, it is also possible to examine the effect that different soft timeout values have on the switch behaviour by modifying the application so that it accepts user supplied values for soft timeout. It is then possible to observe the number of errors caused by flow table overflows (instances of the *All tables full* error) as well as the number of *packets lost*, the latter by observing the difference between the packets transmitted and those received by the target system. By correlating these values, it is possible to determine whether packet loss is occurring due to the flow tables being overfilled or not. A further value of interest is the number of *packet out* (that is, *ofp_packet_out* messages, not total transmitted packets) messages generated by the controller.

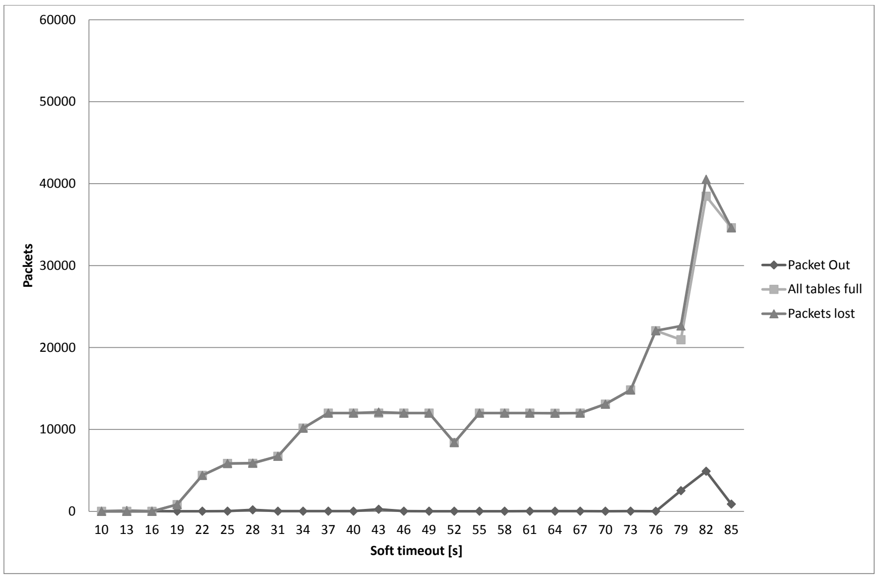


Figure 5.3: Test with data link at 100 Mbps, 10 ms delay, control link at 100 Mbps, 1 ms delay

Figure 5.3 shows a steady increase in both the number of table overflows as well as lost packets with an increasing soft timeout value. There is a long plateau between approximately 37 seconds and 67 seconds. Moreover, for high timeout values, there is also a substantial number of packet out messages generated.
The initial data set was generated using a bandwidth of 100 Mbps for both the control link and the data link, as well as a 10 ms for the data link and a 1 ms delay for the data link. It is unfortunately not possible to accurately simulate bandwidths of 1 Gbps due to limitations of the traffic control facility. However, we may also consider the scenario with the control link has impaired performance. For this, we will use a bandwidth of 10 Mbps and a delay of 10 ms for the control link, with all other values remaining unchanged.

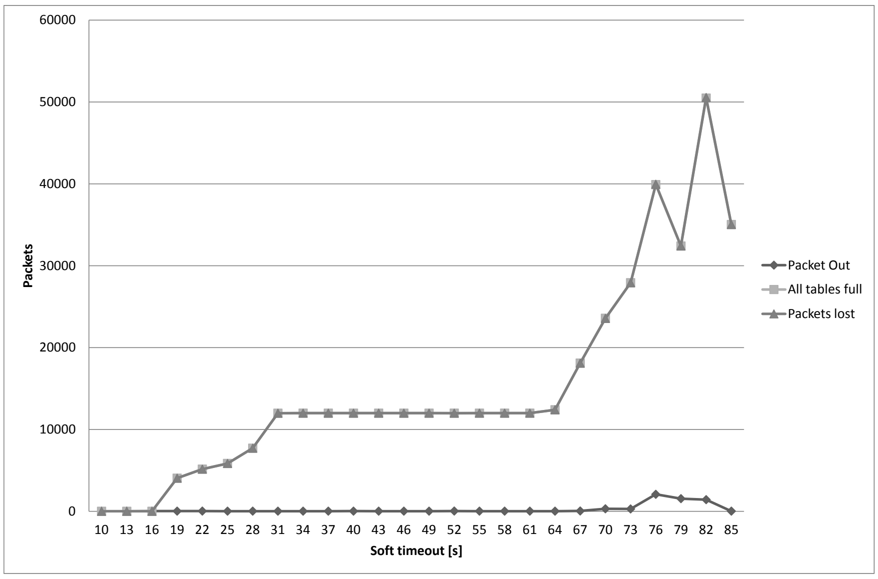


Figure 5.4: Test with data link at 100 Mbps, 10 ms delay, control link at 10 Mbps, 10 ms delay

It may be observed in Figure 5.4 that lower performance on the control link tends to aggravate the effect of denial of service attack, with the plateau of packet loss being reached earlier (about 31 seconds timeout). The packet loss also exceeds this plateau for lower timeout values (about 64 s). There is significantly higher incidence of packet loss and *packet out* messages being sent

for smaller timeout values.
As an alternative to the *forwarding.l2_learning* module, which installs exactly matching flows, we may also use the *forwarding.l2_pairs* module. This module installs flow rules that only match MAC addresses instead. This can be considered to be a crude form of aggregation, which is also proposed as a a mitigation in 6.2.2. At any rate, it is much more scalable than installing exactly matching flow rules, and mimics the functionality of a simple unmanaged layer 2 switch. Using this module, no overflows occur and no packets are lost. The execution times are also shorter.
By adjusting the control link bandwidth, it is also possible to simulate a scenario where the controller-switch link is overloaded; performing this attack from multiple host nodes should have a similar effect. It was not possible to create these conditions on the test environment.

5.4.2 Information Disclosure

The scenario tested here is the one described in 3.5.3.1 (*Determining whether a Flow Rule Exists*). For this attack to be performed, it is necessary that a strategy which performs dynamic aggregation of flow rules is in use. In order to achieve this, a POX-based controller application (entitled *forwarding.l3_aggregator_simple*[6]) is implemented. This application simply sets all values not required for an exact match to wildcards - specifically, the values *dl_src*[7], *nw_src*[8], *tp_src*[9] and *in_port*[10]. In general, forwarding behaviour does not need to depend on ingress filtering, only its destination, so the aggregation of these values is a realistic scenario. In addition to this, as it has been seen above, the more specifically a flow space is defined, the more prone the system is to denial of service attacks.
The aim of the attack is to exploit the use of aggregation in order to discover some aspect of network state that would otherwise not be visible to an attacker. In this case, there are two sets of clients connected via a pair of switches, which are connected together[11]. If a server is connected to the second switch, and several clients to the first switch, then the aggregation occurring in the first switch in response to several connections from the clients to the server could allow another client to deduce that such a connection is in existence. It does this by detecting a difference in the time required for a TCP connection to be established; if a second connection attempt is substantially different (i.e. faster) than the first, then a new flow rule was installed, leading to the conclusion that no connection was in existence. Conversely, if there is little difference, the attacker may conclude that no new flow rule was installed, and therefore a connection was established previously.
We execute this attack several times in order to discover the statistical variance of the measured times. This allows us to derive the degree of certainty with which we may conclude that an existing flow rule is present or not. We also perform the operation with a non-aggregating controller, which uses exact matches[12]. This acts as a control. This can then be compared against the aggregating controller, allowing us to see to what extent the differences in timing are observable and statistically significant. To this end, we will deploy the aggregating controller with symmetrical (10 ms) timing. We will also consider the case where the control path connection is significantly faster, making it more difficult to distinguish the times. In this case, not only the connection latency, but also the performance of the controller are relevant. If the controller - which includes not only the application itself, but the entire network stack, the kernel, the transport layer as well as the software components of the switch[13] - introduces significant delays, we may still be able to distinguish the different cases. The complementary case, where the data path is faster than the control path, is less interesting, as the network latency alone makes distinguishing the different cases relatively straightforward.
Measurements are taken four times in two sets: The first set without aggregation being possible, the second set with the possibility of aggregation[14]. The first measurement of each set is the slow path (before the installation of the flow rule), the second the fast path (after a flow rule has been installed). For each measurement, both the average and the standard deviation are given

[6]Based on the *forwarding.l3_learning* application supplied with the POX source as an example
[7]Link layer source (MAC) address
[8]Network layer source (IP) address
[9]Transport layer source (TCP/UDP) port
[10]Physical or logical port that packet arrived on
[11]The same setup as in Figure 5.2
[12]The same *forwarding.l3_learning* application mentioned above
[13]Especially considering the use of encryption, although that is not being used here
[14]This is achieved by establishing network connections from the neighbouring hosts to the target system

- the latter is important to determine the degree of certainty with which the different cases can be distinguished.

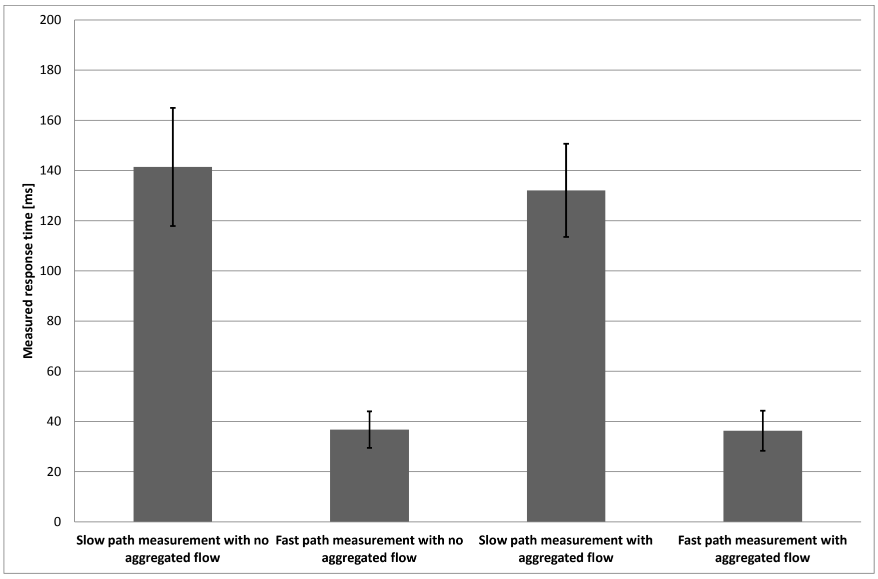


Figure 5.5: Control using *forwarding.l3_learning* controller with symmetric timing at 10 ms

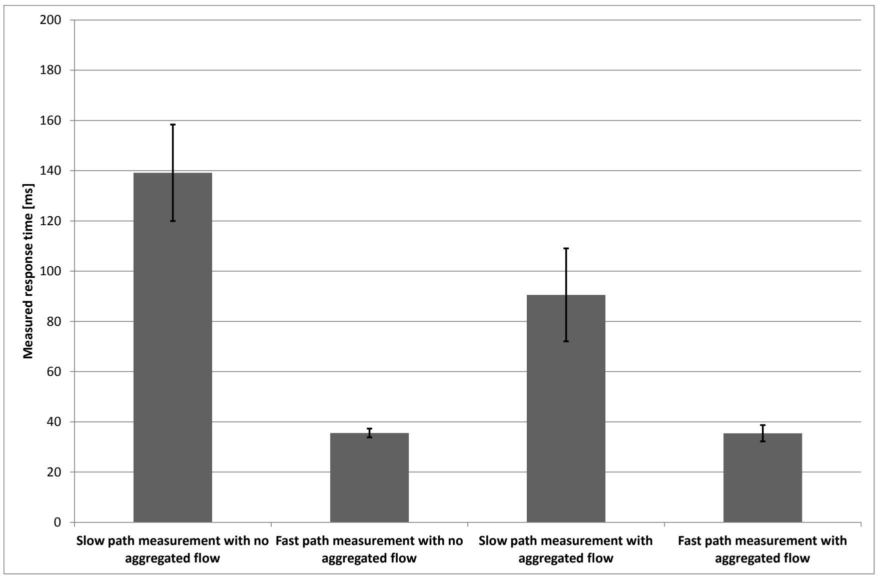


Figure 5.6: Using *forwarding.l3_aggregator_simple* controller with symmetric timing at 10 ms

It is clearly evident that with symmetric timing in Figure 5.6, the difference between the cases - where an aggregated flow rule exists and where none exists - is distinguishable. Compare this to Figure 5.5, where there is little difference between the two times. In the case where asymmetric timing is in effect (Figure 5.7), the difference is much more difficult to distinguish; multiple measurements may be required, and it may not be possible to achieve the desired level of certainty.

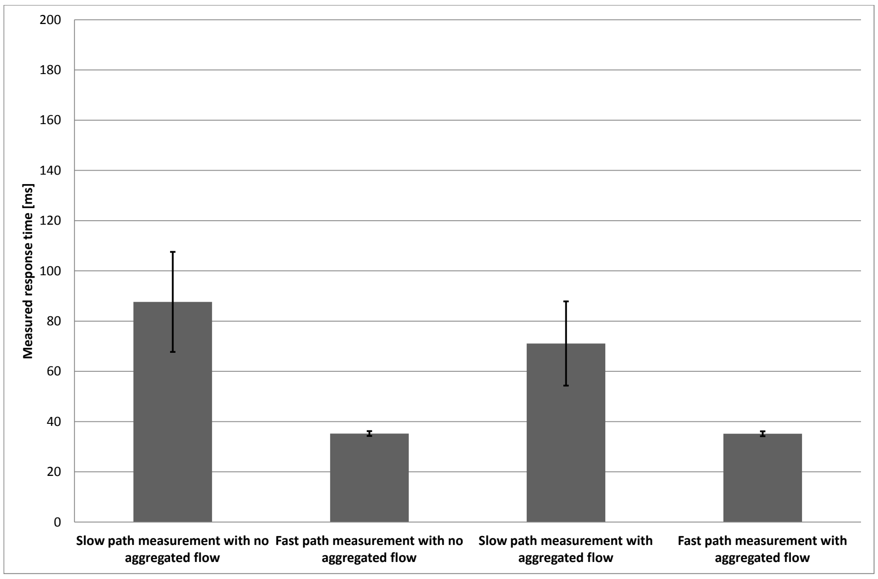


Figure 5.7: Using *forwarding.l3_aggregator_simple* controller with asymmetric timing at 10 ms for data path and 1 ms for control path

We may also consider the *distribution* of the measured times[15]. Figure 5.8 shows a histogram of the control data. The two data sets are the case before a parallel connection is created from another client to the server, and the case afterwards. In the latter case, aggregation may occur, if the controller allows this. In the control, this is *not* allowed, so the distributions should be the same, within tolerance.
We clearly have approximately the same distribution, which is essentially symmetrical and resembles a normal distribution - although we cannot say for sure if the data set is normally distributed or not. Figure 5.9 shows a histogram of measured times when aggregation is actually in effect, for a symmetrically timed link. The distribution of the second data set (with aggregation) is not symmetrical, and the two cases can clearly be distinguished from each other. Compare this to Figure 5.10, where the two distributions are nearly overlapping. Here, a large number of measurements would have to be made in order to distinguish the two cases from each other.

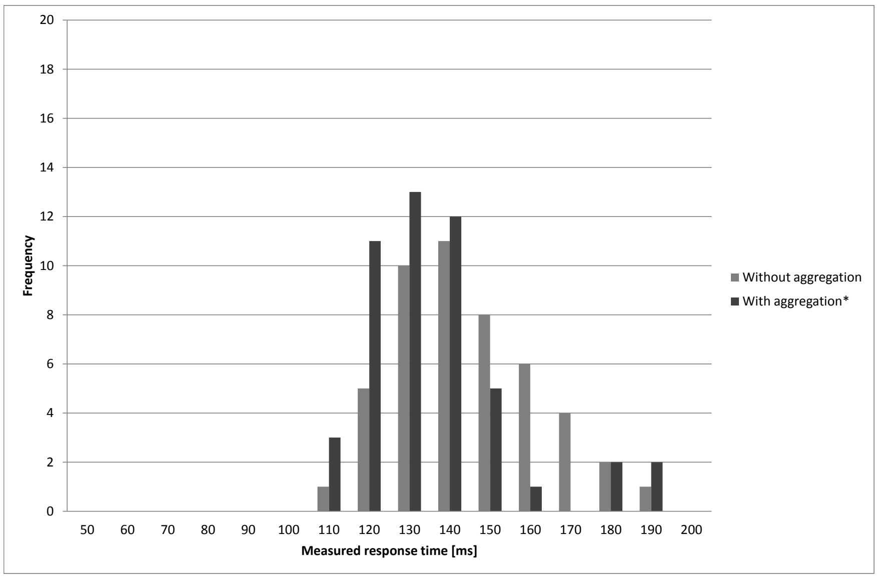


Figure 5.8: Histogram of control using *forwarding.l3_learning* controller with symmetric timing at 10 ms

[15]Each experiment is run 50 times to generate the data.

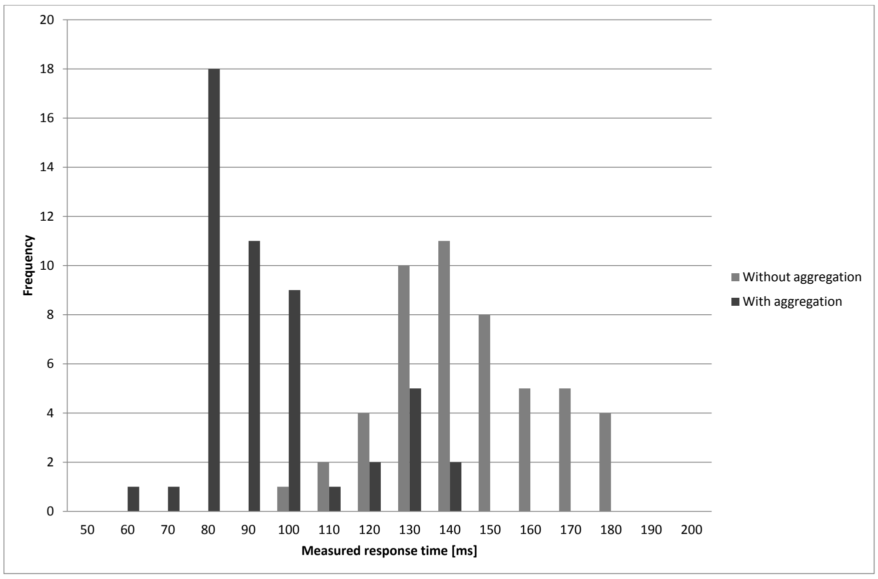


Figure 5.9: Histogram of data using *forwarding.l3_aggregator_simple* controller with symmetric timing at 10 ms

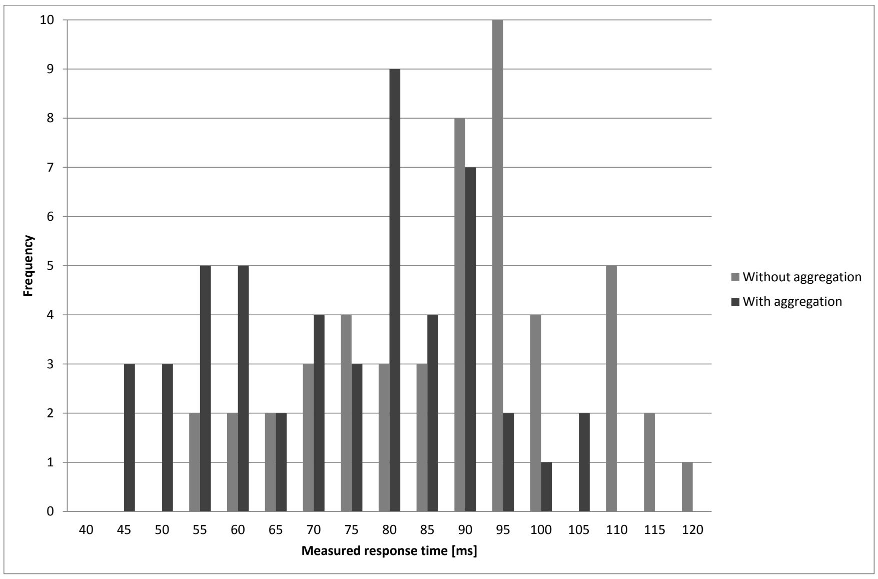


Figure 5.10: Histogram of data using *forwarding.l3_aggregator_simple* controller with asymmetric timing at 10 ms for data path and 1 ms for control path

Chapter 6

Prevention and Mitigation

6.1 Introduction

The question arises of how tampering, information disclosure and denial of service attacks may be prevented or at least mitigated. Firstly, we must ask ourselves whether any such protection must be implemented in the switch, in the controller, in a third party device (such as a firewall) or even in the network architecture itself. Implementation in the switch would require support from the OpenFlow protocol, or at least the use of appropriate extensions. Implementation in the controller would be a more natural solution, conforming to the design of OpenFlow more closely (i.e. that the control logic which regulates the behaviour of the data path resides on the controller not on the switch). It would not eliminate bandwidth or latency related issues, however. We must also consider the various network setups in which OpenFlow may be deployed, as well as the applications for which it is used. The requirements of such a network could be *security* (i.e. preventing external access), *performance* (i.e. throughput and latency) or *reliability* (minimise downtime, fast fail-over in case of link failure). Furthermore, networks may have a known or unknown user base, and that user base may predominately establish flows from inside the (OpenFlow-controlled) network to the outside, or have outside users establish flows into the network. It is also possible that flows are established in both directions in equal measure. Table 6.1[1] shows some types of networks as well as their security requirements, which are described in detail below.

Type	User base known	Flows established	Requirements		
		(main direction)	Security	Performance	Reliability
Corporate	✔	Outbound	↑	→	↑
Academic	✔/✗	Outbound	→	↓	→
Research	✔	Both	↓	↓	↓
Data center	✔/✗	Both	↑	↑	↑
Backbone	✗	Both	↓	↑	↑
DMZ	✗	Inbound	↑	→	↑
Special purpose	✔/✗	Unknown	→	→	→

Table 6.1: Properties of different network types

- A corporate network, with a largely trusted or at least accountable user base, where flows may generally only be established from inside the network to the outside, and not the other way round. Security is important as such networks are usually built with a high expectation of confidentiality.

[1]The symbols ↑, → and ↓ indicate that the property is of greater, equivalent or less importance, relative to other networks.

- An academic network, with a well-defined, but not necessarily trustworthy user base[2]. Access control has a lower emphasis, and users may decide to run ad-hoc servers. It is also possible that users will execute applications which generate very heavy network traffic, for experimental or other purposes.
- A research network, established for the sole purposes of research into network-based applications (Chapter 5 presents an example). The requirements are low, as both traffic and access are controlled by the researcher(s). However, it is necessary for the network to support a wide range of functionality, and be easily programmable. This also applies to networks used for development in a corporate environment. This network environment generally does not present a security risk.
- A backbone network, which handles large amounts of traffic. The user base is unknown and is not considered trustworthy. Network flows may be established in any direction. Performance is most important with little expectation of security, in the sense that the confidentiality and integrity of data being transmitted in the network is not guaranteed. The impact of attacks affecting availability and throughput is very high - these aspects are subsumed under reliability and performance, respectively.
- A data center network, which may have heavy traffic loads but also have multiple pathways and high degrees of redundancy.
- A demilitarised zone (DMZ), which sits on the boundary of a controlled network environment and the Internet. It is essential that systems here are accessible from the outside, and with reasonable performance, but access needs to be controlled for security reasons. It can be considered a special case of a data center network (a DMZ is usually situated in a data center).
- A hermetically isolated, single purpose network with no connection to the Internet at large. Such a network could be a virtual network or a physical one. An example of the latter would be networks used for Internet telephony and television. Research networks would also fall into this category. Security is not a major concern at a network level, although care should be taken in case an unauthorised system connects to the network. These networks may have special requirements or make use of extensions, like virtual network slices or multicasting.

Proposed applications for OpenFlow networks include (with remarks about requirements on controller strategy):

- Switching and routing[3], especially unconventional routing methods[16]. Many of these applications introduce a dynamic element to routing and switching (as opposed to a static forwarding table), making a proactive strategy less effective. OpenFlow can also be used to implement static routing, which can be completely proactive in nature - this is a requirement in backbone networks and desirable for data center networks. In general, all networks require some form of forwarding, and this constitutes the most basic functionality offered by OpenFlow.
- Multicasting and other media distribution methods[40, 33]. The central coordination inherent in the OpenFlow design can make multicast routing considerably easier to implement. Multicasting can be sort of as "subscription" to "published" content (in the case of a common application of multicasting, IPTV, this analogy is quite evident). Since the subscription cannot be predicted in advance, the system is inherently reactive in nature. Although this may be used in some corporate or academic networks, the main use is in backbone networks (where the performance advantages are most notable) and special purpose networks.
- Access control, that is, the application of access control lists to the establishment of new flows[73, 74]. This system is reactive in nature, although a flow rule can be installed to drop traffic that is not permitted. The use of OpenFlow for enforcing security policies

[2]Not trustworthy in the sense that, if there are any security weaknesses, they may be exploited for comparatively benign purposes.

[3]Strictly speaking, the OpenFlow 1.0 specification is not capable of acting as a network layer switch, as it does not support decrementing the TTL field of the IP header.

requires careful consideration: There is a trade-off between granularity and performance. This approach would be used in lieu of a firewall in a corporate network or other network requiring stringent access control. Use on a data center is conceivable, but would require careful consideration of the trade-offs between security and performance. Some form of access control may also be desirable for a DMZ network.

- Load balancing[70]. One example would be to split flows randomly between targets, with the probability of each target being chosen inversely proportional to the current load. This would be applied by the controller, leading to a reactive strategy. With OpenFlow 1.2, this could be implemented proactively, as OpenFlow 1.2 has some in-built support for load balancing. Load balancing would be most applicable to a DMZ or data center network.

- Fail-over and reliability[61, 17, 75]. This functionality does not need to have an influence on controller strategy: It is only necessary for the controller to react if connectivity is lost or degraded (which the controller will learn from a port status message or similar). The controller can then modify the flow table as needed. Newer versions of OpenFlow also support fast fail-over, allowing the switch to change routes without the controller intervening. See also Section 4.4. This practice is most applicable to data center and backbone networks, which actually have multiple paths available; most network types usually use spanning tree-based forwarding as the simplest loop-free strategy.

- Traffic shaping and other peformance-related policy enforcement. As with other dynamic forwarding strategies, the most flexibility is obtained when using a reactive strategy, although (especially with OpenFlow 1.3, see Section 4.4) there is some support in the protocol for traffic shaping through flow meters and QoS, which may be installed in advance with broadly applicable flow rules (assuming multiple flow tables are available). These may be employed in a corporate or academic network to enable performance-sensitive applications (videoconferencing or IP telephony, for instance), or on a data center to enforce traffic policies and ensure that all users receive a certain level of service; this may be stipulated in service-level agreement. The application in backbone networks is highly controversial[4].

- Enforcement of power consumption policy[68]. This highly specific application is proactive in nature, but can be implemented on a system using reactive strategies. The proposed application is focused on data center networks and may also apply to DMZ networks (which are essentially a similar case).

- Network virtualisation and isolation[53, 57, 60]. In the case of FlowVisor, this is actually enforced by an intermediate system between the switch and the controller. In general the presence of virtualisation does not need to dictate a particular controller strategy, but will make the effects of such a strategy harder to predict. Data center networks are of particular interest, but academic and corporate networks are also relevant. Virtualisation is important for research networks, especially when separating these from production networks; this constituted the original purpose of the FlowVisor extension.

- Monitoring and instrumentation[35, 10, 36]. Beyond the counters supplied by the OpenFlow protocol, this could include a controller application (which monitors new flows) or a system receiving mirrored traffic. All types of networks may benefit from more detailed monitoring, but those with heavy, unpredictable traffic are of particular interest: Backbone, data center and DMZ networks, as well as research networks where instrumentation may be a requirement.

6.2 Denial of Service

With the above points in mind, several approaches are conceivable for mitigating denial of service attacks. These are summarised in Table 6.2 and subsequently described in detail.

[4]The use of traffic shaping and similar measures may be considered to violate *network neutrality*

Proposed measures		Implemented on			Suited for
Number	Description	Switch	Controller	Protocol	Network
6.2.1	Rate limiting	✓	✓	✓	All
	Event filtering	✓	✓	✓	
	Packet dropping	✓	✗	✗	
	Reduce timeouts	✗	✓	✗	
6.2.2	Flow aggregation	✗	✓	✗	Backbone Data center DMZ
6.2.3	Attack detection	✓	✓	✗	Corporate Academic DMZ
6.2.4	Access control	✗	✓	✗	Corporate Special cases
6.2.5	Firewall and IPS	✗	✗	✗	Corporate Academic DMZ
6.2.6	Manual intervention	✗	✗	✗	Special cases

Table 6.2: Overview of proposed countermeasures against denial of service attacks

6.2.1 Rate Limiting, Event Filtering, Packet Dropping and Timeout Adjustment

The use of rate limiting is not supported by the OpenFlow 1.0 specification, but support has been added to 1.3.0, together with support for the filtering of asynchronous messages, see Section 4.5. These tools can allow a switch and controller to remain responsive during a denial of service attack. They cannot protect other users from detrimental effects, though, unless it is possible to pinpoint the attacker with sufficient precision that a matching flow rule can be installed. If that were the case, the traffic could be dropped entirely. The use of event filtering may allow certain event types to be handled by a special controller, which may increase system resilience. Timeouts can be shortened to decrease the impact of a denial of service attack, although this will also detrimentally impact other traffic forms[5]. Even under the 1.0 specification, packets may be dropped when the system is unable to cope with the load. It may be possible to make use of quality of service mechanisms in order to selectively drop packets to guarantee continuity of operations even when an attack is in progress. This requires that the network administrators make a positive decision about which services should be prioritised. Examples may include network control traffic, industrial control systems, voice and video traffic[6] and traffic to servers (such as database servers) required for normal operations. Many networks already use QoS features such as *differentiated services*(DiffServ), especially *expedited forwarding* (EF)[15] and *assured forwarding* (AF)[24, 22]. EF can be used to ensure that delay-sensitive traffic is transmitted within time constraints. AF can be used to ensure that some minimum bandwidth is provided for required services; moreover it can also limit traffic. The latter functionality can be provided by OpenFlow itself, but the use of DiffServ also allows it to be used at the boundary of the network by systems not controlled by OpenFlow, as well as between networks not centrally administered.

Any such solution requires that denial of service attacks be either anticipated, or detected and reacted to. This can be done either automatically or manually. These approaches can be effective for any type of network. Though some of them require changes to the switch, these have all been standardised by the Open Networking Foundation already.

The controller needs to recognise that the switch is under attack in order to put countermeasures

[5]See Figure 5.3 on page 53 and Figure 5.4 on page 53 for an illustration of this.

[6]Especially if IP telephony is in use.

into effect, unless they are permanently in place. Identifying the source of the attack is not necessary, and probably not feasible, given that IP and/or MAC spoofing may be in effect. The recognition would instead recognise that a certain acceptable level of traffic has been exceeded and react accordingly. This is discussed in 6.2.3.

6.2.2 Flow Aggregation

Flow aggregation is a controller strategy where one flow rule matches multiple network flows. This can help to reduce the number of flow rules required to match network traffic. The reduction in flow rules comes at the expense of precision. It also introduces the possibility of unintended consequences - aggregated flow rules would also affect users of other systems - if the aggregation is performed automatically[7]. For networks that do not require precise control of flows, this may be an effective approach. The enforcement of security policies would be more limited, but this would have no impact if these were not enforced in the OpenFlow network itself - making this approach attractive for backbone networks and others which have heavy traffic. A fully proactive approach where flow rules cover all possible traffic would be the logical extreme. That would duplicate the functionality of conventional forwarding tables, but allow more flexibility in terms of matching capability and allowing routing to be performed from a single centralised system. This approach notably does not require any changes to be made to the OpenFlow protocol or switch and can be implemented solely on the controller. It also does not require any sophisticated detection heuristics, although setting up the flow table in a proactive manner is more advanced than a simpler reactive strategy. The switch can be configured to drop packets by default rather than forwarding them to the controller. The extent to which the issue of denial of service attacks is mitigated depends on the degree of aggregation. We might define this as the number of effective new flows (a flow being the tuple of source address, destination address, source port, destination port and transport protocol) per installed flow rule. The higher the value is, the closer the system is to a proactive strategy and the less vulnerable it is to denial of service attack against the flow table, input buffer or controller. A fully proactive strategy, where packets are never forwarded to the controller, is immune to this type of attack.

6.2.3 Attack Detection

Automatic detection of denial of service attacks (for instance, through anomaly detection) is an area that has received considerable attention and remains a major research topic regardless of its impact on software defined networking - for examples, see [32, 54, 64, 12]. Such a detection algorithm could at the least establish that an attack is in progress, perhaps allowing the controller to configure the switch to reduce the impact via rate limiting, reduction of soft timeout or prioritisation of certain traffic classes (all measures discussed in 6.2.1). System administrators could also be notified. A more advanced solution could allow such attacks to be characterised, in order to discover their origin and possibly to allow more precise measures to be taken to reduce their impact, such as barring traffic meeting certain criteria. This may also be of use in the event that involvement of law enforcement agencies is desired, allowing incriminating traffic to be sent to a recording system, perhaps a honey-pot. This approach does not counteract an attack by itself, but may be part of an automated security system that is used to direct forwarding behaviour.

Basic detection functionality could be implemented as a controller application - essentially an IPS application for the controller. The processing could be done in a different thread or process in an asynchronous manner. The flexible forwarding behaviour of OpenFlow would allow certain categories of flows to be duplicated to monitoring systems, without interrupting the flow. Although this is possible with existing switching solutions, with OpenFlow this can be standardised and made dynamic, with the monitoring system essentially subscribing to traffic that it wants to receive. These approaches could be combined by selectively duplicating flows deemed to be "high risk" to an external monitoring system. The addition of OXM in 1.2 (see 4.4.2) makes it possible that appropriate extensions could implement deep packet inspection at a relatively low performance cost. DPI would make the detection of malicious traffic considerably more feasible. Even then, detection solutions could already be implemented on the basis of the 1.0 specification - many make use only of network and transport layer headers.

[7]A rudimentary form of aggregation is demonstrated in 5.4.1.

6.2.4 Whitelisting and Access Control

A more drastic solution is the enforcement of an access control list, a “whitelist” approach, which would only allow traffic to and from permitted addresses and with permitted service types. This would require that *all* intended uses be foreseen, at least for in-bound traffic. Traffic originating from inside the trusted domain may be allowed to pass, similar to existing firewall solutions. Whitelisting does not require special functionality on the part of the switch, nor does it require any changes to the protocol. This solution is worth considering for corporate or academic networks where most flows are likely to be instantiated either from internal addresses or from trusted external ones. This is also analogue to NAT solutions in widespread use[8], which also allow outbound traffic and related inbound traffic to pass freely, without allowing connection establishment from the outside. This solution cannot be applied if outside access is required, which is most likely to be the case in a demilitarised zone (DMZ). It is clearly not viable for backbone networks.

6.2.5 Firewall and IPS

Insofar as any attack originates outside a corporate or academic network, a firewall or IPS may be of use to detect and filter traffic. These systems do not separate the user and control path and therefore should be more robust under load, and they are already in widespread use, being needed to protect client systems from attacks. Such systems could also be used to enforce access control rules, although this reduces the utility of OpenFlow, which would no longer be solely in control of the network. The problem of detecting malicious traffic would still remain.

6.2.6 Manual Intervention

It should not be forgotten that, although these approaches function without any kind of manual intervention, many important networks are managed by network administrators; some networks are even under constant surveillance. If this is the case, it may be worth taking into consideration that a network administrator can decide on an appropriate reaction more effectively than an algorithm can, albeit far more slowly. Manual intervention is most useful on controlled networks, especially those which are under continuous surveillance, and against attacks that continue for a substantial period of time.

6.2.7 Hash Collision Prevention

This only applies to the very specific attack described in 3.5.4.1 involving hash collisions to degrade performance. A possible approach to preventing this form of attack is to introduce randomisation in some elements of the flow table entries, making them harder to predict and therefore making it more difficult to find flow rules which collide. It also makes it impossible to attempt to simulate flow rule creation on the attacker’s system. The *cookie* element is ideal for this purpose, since it is not required that its values be sequential. However, care must be taken, as the OpenFlow specification notes that the value need not be stored in hardware; therefore the cookie may have no influence on hashes used for hardware (TCAM) flow tables. Another possibility is to use varying values for timeouts. This may have the further advantage of making the switch behaviour less straightforward to predict. It is unfortunately also possible that the switch may not store timeout values in hardware either. If this is the case, an additional countermeasure could be to simulate the switch hash function on the controller, allowing it to detect hash collisions and take appropriate countermeasures, such as (possibly temporarily) refusing to install offending flow rules. Also, as above, the use of a proactive strategy prevents this form of attack.

6.3 Information Disclosure

Information disclosure, arising from timing analysis, can reveal certain aspects of a network’s state as well as controller strategy to an attacker. The aim of any prevention or mitigation strategy is to ensure that any observable parameters of the system’s operation do not depend

[8]Of course, OpenFlow can itself be used to implement NAT

on the internal system state, apart from those which are observable from the normal operation of the system. For instance, it may be observed whether or not a packet is forwarded; this reveals some (small) information about the flow table and/or the controller's strategy. It is not possible to prevent this from being revealed, nor is this a weakness of the OpenFlow protocol (any system that enforces access to a resource can be probed in this way). However, if it is possible to determine whether (for instance) a new flow rule is being installed in response to a packet that the user has sent, some information is revealed that is not required by the system design, and not available to an attack against a more conventional system. There are several potential approaches to mitigating this issue. The suggestions made here are similar to proposed mitigations of side channel attacks in cryptographic systems[67, 52].

6.3.1 Proactive Strategy

As with 6.2.2, the use of a proactive strategy can prevent the occurrence of an issue by removing the dependency of response time on the network state. This requires a fully proactive strategy to be effective. The use of flow aggregation, where one flow rule matches multiple flows but the flow rule is still installed in response to user-generated traffic, may actually make the situation worse, especially if source address aggregation[9] is in use, as users might be able to determine whether another user has been communicating with a given host.

6.3.2 Randomisation

By increasing the variance of measurable response times, any timing analysis technique may be hindered. This strategy has been proposed to deal with timing attacks against cryptographic systems such as RSA or AES. By increasing the statistical uncertainty, the number of samples required in order to make assumptions about the system can be increased. This may make the attack less feasible, and the obtained results less certain, although there will be a certain performance penalty for introducing delays. Introducing random delays on the data path is far from straightforward; most hardware would not support this and the performance degradation may be unacceptable.

6.3.3 Attack Detection

In analogy to 6.2.3, it may be possible to develop a controller application which detects suspicious traffic. Any attack based on timing analysis (or any other form of side channel attack) is likely to exhibit a distinctive, repetitive traffic pattern, which may be used by the controller to enact countermeasures or to notify an administrator. A response could include dropping suspicious traffic, introducing randomisation or changing strategy.

6.3.4 Enforced Equal Response Time

The use of equal response time for packets travelling over the switch and those being sent to the controller is obviously not desirable for performance reasons. This approach therefore particularly applies to attacks based on the difference in performance between the fast and the slow path of the switch - the part that is implemented in hardware and that which is implemented in software. Making the software component as fast as the hardware component is futile, as the hardware component can match header fields in parallel using TCAM and forward them to the output queue directly. It might be possible, however, to introduce a delay on the fast path to mask performance differences. This would only need to be small, and would not necessarily constitute a bottleneck. It would also be necessary to introduce a stochastic element, as the software path is generally less predictable than the hardware component. This would require changes to hardware and therefore may not be cost-effective, given the limited scope of this form of attack.

[9]That is, traffic from a range of source addresses to a destination (possibly also a range) is covered by one flow rule, which is cheaper in terms of flow table capacity if all packets would use the same port.

6.4 Tampering

The tampering attacks described in 3.5.2 mostly revolve around a sort of cache poisoning, where the controller installs flow rules based on untrustworthy traffic. Due to the architecture of both the IPv4 and IPv6 protocols, this untrustworthiness is inherent; the only way to prevent it would be to introduce security mechanisms into DHCP, ARP, NDP and related protocols, which is certainly not feasible due to extensive requirements for backward compatibility. This issue is not caused by the OpenFlow protocol specifically, but the protocol allows such attacks to have more severe effects, specifically due to the use of a reactive controller strategy. For this reason, all of the approaches described here involve only mitigating the issue to the point where it is no more serious than the corresponding problems taking place within conventional networks.

6.4.1 Proactive Strategy

As with denial of service and information disclosure vulnerabilities, a proactive strategy here also alleviates the issue. This is due to the controller installing flow rules based on untrusted input. A proactive strategy would emulate the functionality of a traditional switch; although the issue is not *prevented* by such an approach, it is no more vulnerable than a regular network. A partially proactive strategy would partially mitigate the issue, depending on the form of aggregation that is in use.

6.4.2 Timeouts

If flow rules are installed which have the effect of diverting traffic to a malicious user, then shortening the timeout will have the effect of reducing the impact of the attack. This will force an attacker to generate a larger number of forged packets, making the attack more likely to be detected. If the timeout is unpredictable, an attacker will be unable to determine when a flow rule is removed due to timeout, again making this form of attack harder to execute and less effective.

6.4.3 Integrity Checking

A controller may also perform sanity checking on received ARP or NDP packets. Simple tests could be to check for conflicting ARP or NDP replies; under normal circumstances unicast IP addresses should never be shared within their scope, so having multiple systems answer to a single request must be considered suspicious. The same principle applies to MAC addresses, which by definition should be globally unique[10], and so cannot appear on more than one port, unless the network has multiple paths. The controller may have some information about network structure, for instance that a certain port is connected only to one subnet, or that a port is connected to single system, so that multiple IP addresses or MAC addresses cannot become assigned to it.

6.4.4 Access Control

An access control solution would involve the controller knowing the entire network topology and only permitting the installation of flow rules which conform to policy. IP addresses must be mapped to permitted switch ports[11]. This solution is sound for a network which is constant in composition, but is inconvenient if the network structure is likely to change. It does not scale to backbone networks, but it may be of interest in corporate networks and other networks which are centrally administered.

[10]This may not apply to administrator-assigned addresses, but then the administrator is responsible for ensuring uniqueness within a broadcast domain.

[11]*Not* MAC addresses, as these can trivially be forged as well

Chapter 7

OpenFlow Extensions

7.1 Introduction

In this chapter, a security analysis of a sample OpenFlow extension will be performed in order to demonstrate the applicability of our methodology to a wide spectrum of OpenFlow-based setups and architectures. The extension for which an analysis will be performed is *FlowVisor*[55, 56], which is among the most widely used extensions and presents interesting questions regarding security. This chapter is structured as follows: Firstly, the architecture of FlowVisor is described in Section 7.2. Then a dataflow model is introduced in Section 7.3, from which vulnerabilities are derived in Section 7.4, as per the STRIDE methodology. This essentially mirrors the structure of Chapter 3, without the use of attack trees.

7.2 Architecture

FlowVisor is a network virtualisation solution based on OpenFlow. It was developed at Stanford University with the intent of allowing the use of virtual network testbeds over real physical networks. In order to ensure that experimental traffic does not interfere with the regular operation of the network, it is necessary to enforce a strict separation of virtual networks. These virtual networks are defined as *network slices*, that is, specific sets of network resources. For the purposes of FlowVisor, a slice is defined as a set of flows running on a topology of switches, which are isolated but may overlap if this is desired. The implementation of network slices is via a system analogue to virtual machines such as VirtualBox or VMWare: The switches, running OpenFlow are connected to network hypervisors, which are in turn connected to controllers. The network hypervisors enforce the separation of network slices so that *each controller only sees its own network slice.* From the controller's point of view, the network slice behaves like a real network; the hypervisor layer is transparent. Neither the switch nor the controller requires any special software support for FlowVisor to function. FlowVisor receives OpenFlow messages from the controllers (controller-to-switch messages) and switches (asynchronous messages), and may either pass them through directly, modify them, or reject them, transmitting an error message back to the sending controller. It is possible to have read-only access to a slice. This may be useful if a controller is intended to supervise traffic only. In this case, the controller receives messages but may not install new flows. Due to the transparency in the design, it is also possible to chain instances of FlowVisor: An instance of FlowVisor may manage a network that is itself virtualised by another instance of FlowVisor, allowing hierarchical stacking of FlowVisor instances. FlowVisor enforces the following types of isolation:

- Bandwidth isolation, implemented by setting VLAN tags on packets. Future OpenFlow specifications may allow more fine-grained control.
- Topology isolation, implemented by FlowVisor denying the establishment of connections not inside a controller's slice. Furthermore, switch ports which are not in the slice are filtered out, while Link Layer Discovery Protocol (LLDP) messages receive special handling.
- Switch CPU isolation is achieved with several methods, including installing short duration drop flow rules to limit the number of packet in messages, rate limiting control messages

to the switch and preventing the controller from installing slow path flow rules (see 3.4.2.3 and 3.5.4.3), using packet out messages to forward the packets directly instead.

- Flow space isolation, implemented by dynamically rewriting installed flow rules, or rejecting them if this is not possible.
- Flow entries isolation[1], implemented by maintaining a per-slice counter of flow rules.
- Control isolation, implemented by rewriting control messages and replacing buffer IDs[2], so that each controller can only reference buffered packets in its own network slice, and so that transaction IDs do not overlap between controllers.

Figure 7.1 describes the operation of FlowVisor: The guest controllers (1), each of which controls its own network slice, send controller-to-switch (3) and receive asynchronous messages (4) from FlowVisor. FlowVisor forwards controller-to-switch to the appropriate switches, and forwards asynchronous messages from the switches to the guest controllers using the defined slice policy (2).

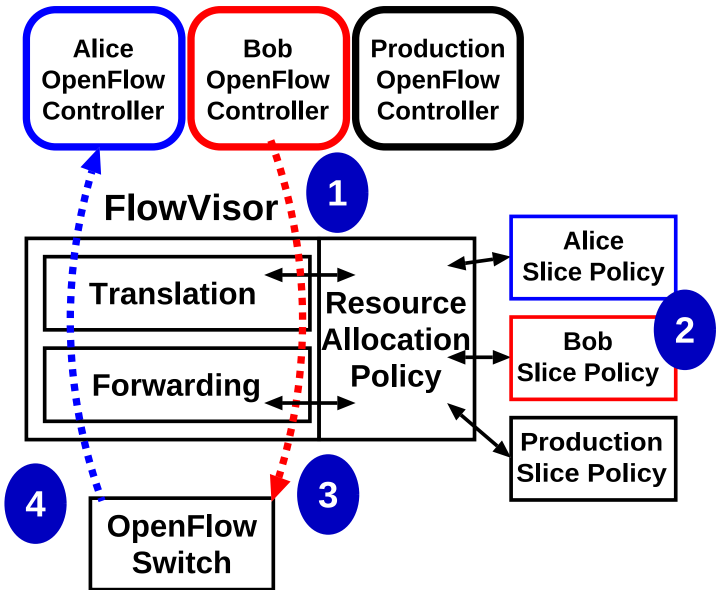


Figure 7.1: Structure of FlowVisor with guest controllers. From [56].

7.3 Data Flow Model

The model for the system is an evolution of the model described in Section 3.3. New in this model is the *FlowVisor* process, which sits between the switch and the controller, and is separated from each by a machine boundary. Asynchronous messages and controller-to-switch messages are passed through the FlowVisor element. The element itself is structured as follows: There is a *translation and forwarding* unit (process). These perform the necessary changes to control packets, and forward them to the appropriate controller or switch. There is a *resource allocation* module (also a process), which makes use of a *slice policy* (data store) in order to control the slicing rules. The translation and forwarding module requires some state information to be stored, in a *network state* data store. There are also buffers on either side of the translation and forwarding unit. The *administrator* has access to the slice policy through an *administrative interface*. A trust boundary exists between the slice policy and the resource allocation module, although it is only possible for the allocation module to read policy and not write it. For the sake of completeness, a boundary also exists between the administrator and the administrative interface, although this has no security relevance under the assumptions made in Section 3.3, which will be preserved here.

[1]This is to prevent one slice from filling the flow table itself
[2]The mechanism is somewhat analogous to the port translation performed in a NAT switch.

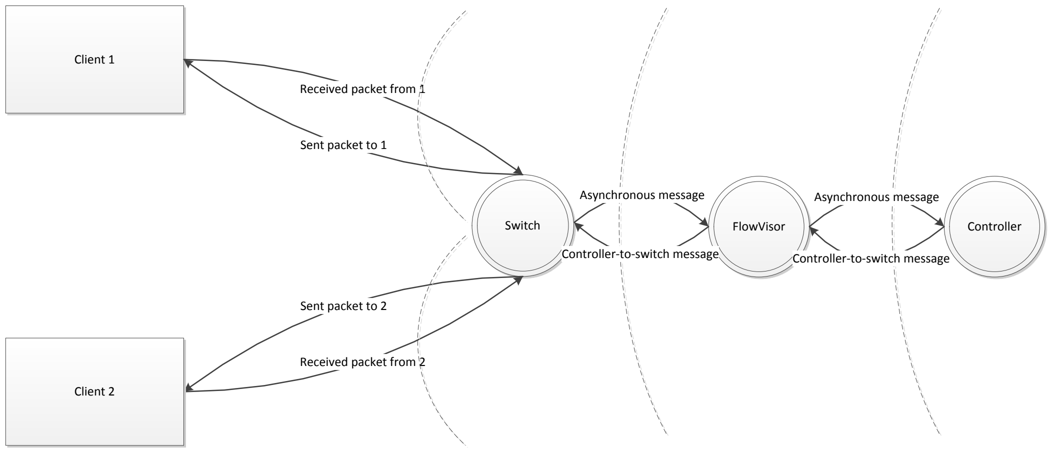


Figure 7.2: Data flow diagram of system including FlowVisor

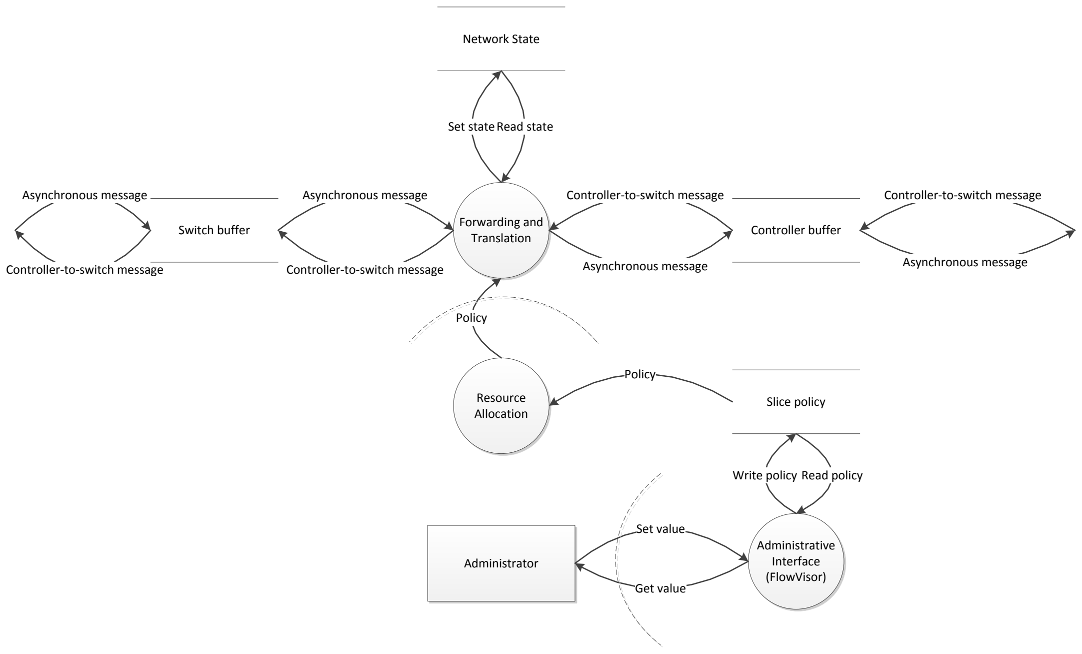


Figure 7.3: Data flow diagram of FlowVisor

7.4 Vulnerabilities

In this section, the potential vulnerabilities of the system according to the STRIDE model will be discussed, analogue to Section 3.4. The same restrictions apply here that also apply to the analysis of OpenFlow itself: The security of interactors will not be discussed, and the administrative components are considered secure, the assumption being that there is a separate control network. Data flows are considered security relevant only when they cross a trust or machine boundary. The data flow *policy* flows from a more to a less trusted segment of the system, and therefore will not be considered here. The data flow(s) *asynchronous message/controller-to-switch message* will be considered in light of the possibility of information disclosure through side channel attacks.

7.4.1 Data Stores

7.4.1.1 Buffers

The system requires buffers on either side in order to ensure that asynchronous processing is performed. The model above (with a unified input/output buffer on either side) is one of several possible means in which this may be represented. As this part of the system design is inferred and not part of the specification, the exact implementation is not relevant and care should be taken to ensure that alternative system designs are taken into consideration. The issue which is of most concern here is a *denial of service* attack, which is directly analogous to attacks described in 3.4.1.1 (*Input Buffer*) and 3.4.1.2 (*Output Buffer*). Such attack would take the same form as the attack described in those sections and is subject to the same conditions. There is no reason to believe that FlowVisor is more vulnerable to this form of attack than a controller, however given the nature of FlowVisor as a virtualisation intermediary, it is important to ensure that such an attack against one virtualised network does not excessively effect the operation of other network slices. *Tampering* is not an issue, as the *Forwarding and Translation* process is responsible for enforcing network separation (which would be the security property that a tampering attack would target) - such an issue would arise in the process *Forwarding and Translation* instead. *Information disclosure* from the buffers may be possible in that the buffer load may affect response times for other network slices, and an overloaded buffer could result in control message loss (depending on how this is implemented), which might be detected on other network slices[3].

7.4.1.2 Network State

This data store is also inferred rather than explicitly described by the FlowVisor specification. This contains mappings of transaction[4] and buffer[5] IDs, port status and the like. As this data store contains data generated by user-initiated processes, it may be vulnerable to *denial of service* attacks. This would take the same form as the attacks described in 3.4.1.3 (*Flow Table*). It seems unlikely that a hash collision attack would be feasible, given the lack of information about which controller is responsible for the corresponding network slice. Due to the nature of the data store, specifically its usage as internal storage by the Forwarding and Translation process, *information disclosure* via side channel attacks is not a concern. A denial of service attack might reveal some information about the network state of neighbouring network slice. By considering the degree of performance degradation and the quantity of data necessary to fill the data store, it may be possible to determine how much space is used by other slices. From this, some inferences about other slices might be made[6], but the threat is marginal. For *tampering* issues, see 7.4.2.1 (Forwarding and Translation).

7.4.1.3 Slice Policy

This data store is described in [56]. It contains the information used to match incoming messages from the switches and controllers to the appropriate network slices. This data store is read-only

[3]Flow rules would no longer be installed

[4]*Transaction IDs* are used to uniquely identify control messages

[5]*Buffer IDs* are used to forward packets stored in the switch's input buffer, to avoid forwarding the entire packet contents back through the control channel

[6]If the load on neighbouring slices is high, an attacker may observe that the data store fills more quickly

from the perspective of the external network; it may only be modified by the administrator. It is therefore not vulnerable to either *tampering* or *denial of service*. *Information disclosure* may be accomplished indirectly via the data flow *Asynchronous Message/Controller-to-switch Message.*

7.4.2 Processes

7.4.2.1 Forwarding and Translation

This process has two functions, as the name suggests: Firstly, received messages, allocated to a particular network slice, are *translated* in such a way that the function of FlowVisor is transparent to both the switches and the controllers. Secondly, the packets are *forwarded* onto the switch or controller to which the slice belongs. *Denial of service* attacks may have the objective of overwhelming the forwarding/translation process, of filling available memory (see 7.4.1.1 and 7.4.1.2), or of exploiting any security holes in the process itself. It is essential that isolation is ensured, therefore a denial of service attack on one network slice should have no effect on any other slice. Based on the design outlined in [56], it is not clear how this can be guaranteed. *Tampering* should be taken into consideration by the administrator. It must be ensured that it is not possible to inject packets into another network slice, though it is assumed that a reasonable allocation policy will prevent this from occurring in practice. With respect to *information disclosure*, a potential issue is that the performance of one network slice may impact the performance of other network slices in a way that is detectable and measurable; this may be used to learn about other network slices, violating the isolation. In particular, it may be possible to learn the network load on another slice by measuring throughput and response times.
In addition to this, the operation of the virtualisation itself may cause some issues. For instance, flow rules which are executed on the switch's processor are generally not installed on the switch itself, but rather handled by FlowVisor directly[7], which is far more time consuming. If a user were to notice this, he or she may infer that network virtualisation is in use. Moreover, the extra network traffic may present a possibility for a denial of service attack. The enforcement of bandwidth and CPU time limits by FlowVisor is also rather rudimentary; this is due to limitations in the OpenFlow protocol rather than shortcomings of the FlowVisor design and it is likely that isolation will be improved when newer versions of the OpenFlow protocol enter into widespread use.

7.4.2.2 Resource Allocation

This process represents the modular component responsible for resource allocation: It defines how packets are mapped to network slices. In particular, its modular nature allows other criteria to be used for partitioning the network into slices than those supported by default. As it stands, the slicing policy is not itself vulnerable to attack, for the same reasons as discussed in 7.4.1.3 (*Slice Policy*). A more complicated module might open avenues of attack: If, for instance, the assignment of network traffic to slices was performed dynamically rather than statically, the attack surface would increase significantly. These may include denial of service attacks targeting the internal state or processing, information disclosure attacks seeking to discover the slicing policy by examining reactions to attacker-generated traffic, or tampering attacks attempting to influence the slicing policy. The exact form of these attacks would depend on the implementation of the module, however possible attacks must be taken into account if a new module is to be implemented.

7.4.3 Data Flows

7.4.3.1 Asynchronous Message/Controller-to-switch Message

This represents several data flows through the "data path" of FlowVisor[8]. Due to the time delay introduced by the presence of FlowVisor, the possibility of *information disclosure* due to a timing analysis attack is introduced. This is analogous to the attack described in 3.4.3.3 (*Asynchronous Message*). Due to the presence of multiple network hops, more information can be obtained here than in the case of an ordinary OpenFlow network. If the timing properties of different

[7]"*FlowVisor prevents guest controllers from inserting slow-path forwarding rules by rewriting them as one-time packet forwarding events, i.e., an OpenFlow "packet out" message.*" from p.7 of [56]

[8]Which is actually the *control path* of the OpenFlow network.

OpenFlow controllers is different, and this difference is distinguishable, it may even be possible to detect which controller is attached to which network slice.

7.5 Conclusion

FlowVisor promises to allow the operation of virtual network slices over production networks. With this in mind, it is important to consider the consequences of its operation. The extra complexity and overhead of FlowVisor increases attack surface area, as compared to an ordinary OpenFlow network. The FlowVisor system is a single point of failure in case of a denial of service attack, which could also prevent the operation of the production network. Given timing differences between a FlowVisor network and an ordinary OpenFlow network, the slicing may be noticeable and certain aspects of its operation observable. An attacker may even be able to determine which controller is managing a particular network slice, allowing him or her to make inferences about slicing policy. The slicing policy itself must be checked carefully to ensure that it is self-consistent; a form of validation may be useful here. It is not clear that all of the isolation properties proposed in [56] can be enforced in practice. Some may even aggravate security issues, by increasing the network traffic load, making it more vulnerable to denial of service attacks.
The exploitation of the denial of service and information disclosure issues here take similar forms to those considered in Section 3.5, so no separate attack trees are provided for these issues. Some issues here may be alleviated by applying newer versions of OpenFlow - this applies mostly to the enforcement of isolation properties. The introduction of bandwidth metering in 1.3.0 (Section 4.5) greatly increases the efficacy of this form of isolation. It also provides an avenue to protect from denial of service attacks. In general, FlowVisor constitutes an interesting tool, but it should be used with care on production networks with untrusted users.

Chapter 8

Future Work

8.1 Introduction

As stated in Section 1.2, OpenFlow is no longer merely of academic interest, as it is increasingly being deployed in real-world systems[26]. It has often been the case that technical standards have been focused on functionality at the expense of security. In the future, security should be considered from the very first draft, not added on shortly before release or offered as an extension.

8.2 Security Modelling

This thesis makes use of two different methodologies - described in 2.2.1 (*Uncover Security Design Flaws Using The STRIDE Approach*) and 2.3.1 (*Threat Modelling Using Attack Trees*) - to model and analyse the OpenFlow protocol. The STRIDE methodology is dataflow-oriented and does not take into account other aspects of security, such as timing. Attack trees require that potential attacks already be known. Numerous other methods described in Section 2.4 may be attempted, although the protocol as described in the standards unfortunately can not be readily modelled using state based models such as Petri nets. Even so, for the sake of validity it would be useful to compare the chosen methodologies with other existing methodologies.
It would also be desirable to apply the STRIDE method to newer OpenFlow standards. This might have been attempted in this paper, however the newer standards have seen few implementations so far, making the 1.0 standard by far the most relevant in practice. Chapter 4 gives a brief overview of the potentially security-relevant changes introduced in the subsequent OpenFlow standards. These changes have not been modelled or analysed using a formal methodology of any kind; this or the modelling of other OpenFlow extensions could be the subject of a future work, if these see widespread adoption.

8.3 Empirical Testing

This thesis makes use of Mininet 2.0 as a test environment for experimentation with OpenFlow. Although the primary intent of Chapter 5 is to attempt to exploit issues described in Chapter 3, it could also be used to model a wide range of other scenarios, as OpenFlow-managed networks can encounter scalability and performance issues in the absence of any malicious traffic. The mitigations proposed in Chapter 6 have yet to be tested, especially on production systems. It would be particularly desirable to have a test suite that could be used to validate switch and/or controller design, as a form of general stress test or as a specific unit test for particular vulnerabilities; the software developed for Chapter 5 could be a basis for this. Testing switch and controller design has also been the topic of other works, such as [11]. The use of physical hardware would allow the impact of such attacks on real-world systems to be determined; there are significant differences in the behaviour of a simulated network environment with little or no concurrent traffic and a real network environment with substantial non-malicious traffic and other performance constraints. Furthermore, the increasing use of virtual networks in cloud systems makes testing attack scenarios on such systems vital - it is known that virtualisation environments allow for novel attack types[4].

Chapter 9

Summary

This thesis consists of a security analysis of OpenFlow, an experimental examination of some the issues uncovered in the analysis and a discussion of strategies to prevent and mitigate security issues found. Here, some of the most important results will be summarised.
The analysis was based on the STRIDE methodology (described in 2.2.1). Based on an analysis of the OpenFlow Switch Specification 1.0, a data flow model was developed, describing the interaction of the switch and the controller. The application of the STRIDE methodology results in an enumeration of *vulnerabilities.* These include *denial of service*, *information disclosure* and *tampering* attacks. The denial of service vulnerabilities arise from the flow table, the control channel between the switch and the controller, and the controller. An attacker generating a large number of flows can overload these components, resulting in the network no longer operating correctly. The information disclosure issues arise from the time delay caused by packets being transmitted over the control channel, as well as delays caused by the fast and slow path of the switch. All of these represent *timing analysis* attacks - a form of *side channel attack.* Tampering issues arise from the installation of flow rules based on packets originating from an untrusted host by a controller. Such packets may include LLDP packets with forged source addresses, leading the controller to install flow rules based on false information. The issue is therefore similar to *cache poisoning attacks*, such as ARP or DNS cache poisoning attacks. A number of other issues are also discussed, from the possibility of counters overflowing, and the consequences this might have, to the possibility of mounting a hash collision attack against various data stores which are implemented as a hash table. Subsequently, there is a discussion of the impact of newer versions of the OpenFlow specification, which both allow some of the security problems of OpenFlow to be mitigated, but also increase its complexity and therefore its attack surface.
The experimental examination performs a demonstration of the denial of service and information disclosure vulnerabilities. Based on setup using the *Mininet* network simulator and the *POX* controller framework, the effects of a denial of service attack are simulated with varying timeout values, as well as different performance constraints. A different controller also emulates the effect of *aggregation* (covering more than one network flow with the same flow rule). The results demonstrate that shorter timeout values as well as greater aggregation decrease the effect of denial of service on network performance. The effects of a denial of service attack are also shown to be dependent on the performance of the link between the controller and the switch, as was to be expected. The information disclosure issue is evaluated with simulated server and several clients. By using a simple aggregating controller - it merely applies wildcards to source address and port values - it is possible to demonstrate that the attacking host can determine whether or not a connection is active between a third party and the server. It is also shown that the latency of the control link has an effect on the ease with which such an inference can be made.
Finally, there is a discussion about prevention and mitigation techniques for the issues described above. Different *network types* as well as different *applications* of OpenFlow and their requirements are reviewed. For denial of service, information disclosure and tampering attacks, several approaches are introduced and their domain of application, advantages and disadvantages discussed.

Nomenclature

actions	The fields of the flow table entry which are used to define the actions the switch should take on packets which match the flow rule
aggregation	The creation of flow rules using wildcards and/or network prefixes so that they encompass multiple network flows
anomaly detection	A method for detecting unusual or anomalous aspects of network flows, in order to detect attacks, misconfiguration, link failure or other issues
asymmetric	Type of communication between the switch and controller. Involves messages that are sent by the switch to the controller without its solicitation and without a reply being waited for, for instance to notify the controller of a new flow
attack trees	A schematic representation of potential methods of attacking a system, based on fault analysis trees
authentication	Security measure to verify that a user or system identity matches the claimed identity. Protects against spoofing
authorization	Security measure to ensure that an authorised user or system is only able to perform actions or access data that it is entitled to, according to policy. Protects against elevation of privilege
availability	Security measure to ensure that a process or system remains available to authorised systems or users, even in the face of malicious interference. Protects against denial of service
confidentiality	Security measure to prevent a user or system from accessing data without prior authentication and authorisation. Protects against information disclosure
control plane	The part of a switch designed to handle the updating of the forwarding table, as well as other operations that do not occur at line rate. In an OpenFlow system, these operations are performed on an external controller
controller-to-switch	Type of communication between the switch and controller. Involves messages that are sent by the controller to the switch without its solicitation, such as that used to install a new flow rule
counters	The fields of the flow table entry which are used to keep track of per-flow statistics
data flows	In a data flow diagram, the element that models a flow of data between two other elements, such as a network connection. Also includes pure internal flows.
data plane	The part of a switch designed to handle the forwarding of packets, as well as other supported operations (such as stripping a VLAN tag), at line rates
data stores	In a data flow diagram, the element that models a store of data, such as a file or database
dataflow diagram	A schematic description of the flows of data inside a system. Contains elements of data stores, data flows, processes and interactors
demilitarised zone	In a network protected by firewalls and/or intrustion preven-

	tion systems, the part of the network which is accessible to external systems, typically containing servers providing services to third parties (web, email, DNS and so forth)
denial of service	Attack type that involves malicious disruption of the operation of a system or process. For instance, a TCP SYN flood or reset attack
differentiated services	A form of quality of service enforcement where traffic is classified by a tag, and traffic with different levels of priority is forwarded in different queues. Depending on the classification, guarantees of performance may be provided. Low priority traffic may be dropped preferentially
drop	Supported action on the switch. Results in packet being abandoned
elevation of privilege	Attack type that acquiring permissions to perform actions and/or access data that a user or process by design is not entitled to. For instance, a stack buffer overflow
enqueue	Supported action on the switch. Same parameters as the forward action, but uses per-port queueing to enforce quality of service
flow	A network flow is a tuple containing the transport layer protocol, the source and destination addresses and the source and destination ports
flow rule	An entry in the flow table, describing the desired reaction to received packets matching specified headers. Installed by the controller.
flow table	Data store on the switch that consists of a list of flow rules installed by the controller. Each rule consists of header fields that must be matched (possibly with wildcards), actions to be taken on match as well as counters for statistical purposes
forward	Supported action on the switch. Results in packet being forwarded to a physical or virtual port
header fields	The fields of the flow table entry which are used for matching the header fields of the packet
information disclosure	Attack type that involves acquiring information that by design a user or process should not have access to
integrity	Security measure to verify that data has not been maliciously modified. Protects against tampering
interactors	In a data flow diagram, the element which represents external systems (including people) that interact with the system components
LLDP	Short for Link Layer Discovery Protocol. Protocol which supports the discovery of network topology and of the capabilities of devices connected to the network
management network	The network that is used to connect the controller to switches under its control
Mininet	A network virtualisation testbed which can simulate arbitrary network topologies as well as a variety of performance constraints. Based on Linux network namespaces and implemented with OpenFlow, it has a Python-based API and can also be used from the command line in simple cases.
modify field	Supported action on the switch. Allows a field of the packet header to be modified as specified in the flow table
NDP	Short for neighbourhood discovery protocol, this provide services for IPv6 networks similar to, but more extensive than, the services that ARP provides for IPv4 networks
netcat	Networking tool which allows data to be piped over TCP and UDP sockets to another client running netcat
network neutrality	The proposed principle that all types of traffic in a network

	should be treated equally, generally precluding the use of traffic shaping and quality of service measures
network slices	A virtual network that is implemented by a network hypervisor, such as that implemented in FlowVisor. The slice is similar to a VLAN, but with fewer constraints in its structure. Individual network slices are strictly separated from each other, just as virtual machines running on a physical machines are.
non-repudiation	Security measure to prevent a user or system from credibly denying an action that it has performed, or its authorship of data. Protects against repudiation
POX	A Python-based, modular controller framework. Can support a variety of strategies
proactive strategy	A strategy where the controller anticipates new flows and installs new flow rules before they are instantiated. The flow rules may also aggregate several flows into a single flow rule using wildcards
processes	In a data flow diagram, the element that represents processes, either actual programmatic processes, or abstract processes - which may be broken down into subprocesses
reactive strategy	A strategy where the controller installs flow rules in response to received packets instantiating new flows
repudiation	Attack type that involves denying the origin of data or actions
role based access control	A system of access control where access policy is determined by a user's assigned roles in a system (a user can have any number of different roles)
scapy	A Python-based packet generation framework that can be used to simulate, or actually perform, a variety of network-based attacks.
service-level agreement	A contract defining the minimum level of performance and reliability that must be provided (downtime, throughput, latency, response time etc.)
side channel attack	A class of attacks that result in information disclosure through the external measurement of observable quantities which are correlated with internal system properties, where the system design would not allow the properties to be revealed. Examples include timing analysis, power analysis and acoustic analysis
software defined networking	A networking paradigm where the control plane of a network is separated from the data plane.
spoofing	Attack type that involves emulating another identity. For instance, IP address spoofing
STRIDE	Abbreviation for Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of privilege. Approach to analysing the security of a system design
symmetric	Type of communication between the switch and controller. Involves messages sent from either controller or switch to the other and for which a response is expected, for instance messages generated to ensure connection liveness
tampering	Attack type that involves malicious manipulation of data. A TCP connection hijacking attack is one example of this
timing analysis	A type of side channel attack that results on information disclosure due to the internal (confidential or not otherwise available) state of a system having a predictable and statistically significant effect on measurable response times
traffic analysis	A form of attack which makes use of transmission metadata rather than the contents of (encrypted) transmission
trust boundaries	In a data flow diagram, an element which separates elements at different level of trust, for instance between privileged and unprivileged operations

References

[1] Marwan Abi-Antoun, Daniel Wang, and Peter Torr. Checking threat modeling data flow diagrams for implementation conformance and security. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 393–396, New York, NY, USA, 2007. ACM.

[2] S. Al-Fedaghi and A.A. Alrashed. Threat Risk Modeling. In *Communication Software and Networks, 2010. ICCSN '10. Second International Conference on*, pages 405–411, feb. 2010.

[3] Amenaza Technologies. SecurITree. `http://www.amenaza.com/SS-what_is.php`. Accessed on 02.04.2013.

[4] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, pages 103–108, New York, NY, USA, 2010. ACM.

[5] David Basin, Manuel Clavel, and Marina Egea. A decade of model-driven security. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 1–10, New York, NY, USA, 2011. ACM.

[6] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security for process-oriented systems. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 100–109, New York, NY, USA, 2003. ACM.

[7] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, January 2006.

[8] Udi Ben-Porat, Anat Bremler-Barr, Hanoch Levy, and Bernhard Plattner. On the vulnerability of hardware hash tables to sophisticated attacks. In *Proceedings of the 11th international IFIP TC 6 conference on Networking - Volume Part I*, IFIP'12, pages 135–148, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] Philippe Biondi. Scapy. `http://www.secdev.org/projects/scapy/`. Accessed on 02.04.2013.

[10] R. Braga, E. Mota, and A. Passito. Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 408–415, oct. 2010.

[11] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test openflow applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[12] Jieren Cheng, Jianping Yin, Chengkun Wu, Boyun Zhang, and Yun Liu. DDoS attack detection method based on linear prediction model. In *Proceedings of the 5th international conference on Emerging intelligent computing technology and applications*, ICIC'09, pages 1004–1013, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Cisco Systems. Virtual LAN Security Best Practices. `http://www.cisco.com/warp/public/cc/pd/si/casi/ca6000/prodlit/vlnwp_wp.pdf`. Accessed on 02.04.2013.

[14] K. Daley, R. Larson, and J. Dawkins. A structural framework for modeling multi-stage network attacks. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 5–10, 2002.

[15] B. S. Davie, A. Charny, J. W. R., K. Blair Benson, J. Y. Le, W. Courtney, S. Davari, and V. Firoiu. An expedited forwarding PHB (Per-Hop behavior). RFC 3246, Internet Engineering Task Force, March 2002.

[16] P. Dely, A. Kassler, and N. Bayer. OpenFlow for Wireless Mesh Networks. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6, 31 2011-aug. 4 2011.

[17] M. Desai and T. Nandagopal. Coping with link failures in centralized control plane architectures. In *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, pages 1–10, jan. 2010.

[18] Martin Devera. HTB - Hierarchy Token Bucket. `http://lartc.org/manpages/tc-htb.html`. Accessed on 02.04.2013.

[19] O. El Ariss, Jianfei Wu, and Dianxiang Xu. Towards an Enhanced Design Level Security: Integrating Attack Trees with Statecharts. In *Secure Software Integration and Reliability Improvement (SSIRI), 2011 Fifth International Conference on*, pages 1–10, june 2011.

[20] Omar El Ariss and Dianxiang Xu. Modeling security attacks with statecharts. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 123–132, New York, NY, USA, 2011. ACM.

[21] Giovanni Giacobbi. The GNU Netcat project. `http://netcat.sourceforge.net/`. Accessed on 02.04.2013.

[22] D. Grossman. New terminology and clarifications for diffserv. RFC 3260, Internet Engineering Task Force, April 2002.

[23] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[24] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding PHB group. RFC 2597, Internet Engineering Task Force, June 1999.

[25] Shawn Hernan, Scott Lambert, Tomasz Ostwald, and Adam Shostack. Uncover Security Design Flaws Using The STRIDE Approach, 2006.

[26] Urs Hoelzle. OpenFlow @ Google. `http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf`. Accessed on 02.04.2013.

[27] HP Networking. OpenFlow. `http://h17007.www1.hp.com/us/en/mobile/solutions/enterprise/openflow.html`. Accessed on 02.04.2013.

[28] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 127–132, New York, NY, USA, 2012. ACM.

[29] Juniper Networks. OpenFlow Switch Application (OF-APP) for Juniper MX-Series Routers. `https://developer.juniper.net/shared/jdn/docs/ProgrammableNetworks/OpenFLow_APP_JDN_Overview.pdf`. Accessed on 02.04.2013.

[30] P.A. Khand. System level security modeling using attack trees. In *Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on*, pages 1–6, feb. 2009.

[31] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[32] Seong Soo Kim and A. L. Narasimha Reddy. Statistical techniques for detecting traffic anomalies through packet header data. *IEEE/ACM Trans. Netw.*, 16(3):562–575, June 2008.

[33] Boris Koldehofe, Frank Dürr, Muhammad Adnan Tariq, and Kurt Rothermel. The power of software-defined networking: line-rate content-based routing using OpenFlow. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '12, pages 3:1–3:6, New York, NY, USA, 2012. ACM.

[34] lxc Linux Containers. `http://lxc.sourceforge.net/index.php/about/kernel-namespaces/network/`. Accessed on 02.04.2013.

[35] Arif Mahmud, Rahim Rahmani, and Theo Kanter. Deployment of Flow-Sensors in Internet of Things' Virtualization via OpenFlow. In *Mobile, Ubiquitous, and Intelligent Computing (MUSIC), 2012 Third FTRA International Conference on*, pages 195–200, june 2012.

[36] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. Revisiting traffic anomaly detection using software defined networking. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 161–180, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] Paul Menage. Control Groups. `http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`. Accessed on 02.04.2013.

[38] Mininet. `http://mininet.github.com/`. Accessed on 02.04.2013.

[39] A. Morais, E. Martins, A. Cavalli, and W. Jimenez. Security Protocol Testing Using Attack Trees. In *Computational Science and Engineering, 2009. CSE '09. International Conference on*, volume 2, pages 690–697, aug. 2009.

[40] Yukihiro Nakagawa, Kazuki Hyoudou, and Takeshi Shimizu. A management method of IP multicast in overlay networks using openflow. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 91–96, New York, NY, USA, 2012. ACM.

[41] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 1–9, New York, NY, USA, 2008. ACM.

[42] T. Narten, E. Nordmark, and W. Simpson. Neighbor discovery for IP version 6 (IPv6). RFC 1970, Internet Engineering Task Force, August 1996.

[43] NCSA. MyProxy. `http://grid.ncsa.illinois.edu/myproxy/`. Accessed on 02.04.2013.

[44] NOXRepo.org. About POX. `http://www.noxrepo.org/pox/about-pox/`. Accessed on 02.04.2013.

[45] NOXRepo.org. NOX. `http://www.noxrepo.org/`. Accessed on 02.04.2013.

[46] Open Networking Foundation. OpenFlow-switch. `https://www.opennetworking.org/`. Accessed on 02.04.2013.

[47] Oracle. VirtualBox. `https://www.virtualbox.org/`. Accessed on 02.04.2013.

[48] Phil Porras. www.Openflowsec.org. `http://www.openflowsec.org/`. Accessed on 02.04.2013.

[49] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for OpenFlow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. ACM.

[50] J. B. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[51] Vineet Saini, Qiang Duan, and Vamsi Paruchuri. Threat modeling using attack trees. *J. Comput. Sci. Coll.*, 23(4):124–131, April 2008.

[52] Kazuo Sakiyama, Elke De Mulder, Bart Preneel, and Ingrid Verbauwhede. Side-channel resistant system-level design flow for public-key cryptography. In *Proceedings of the 17th ACM Great Lakes symposium on VLSI*, GLSVLSI '07, pages 144–147, New York, NY, USA, 2007. ACM.

[53] Elio Salvadori, Roberto Doriguzzi Corin, Matteo Gerola, Attilio Broglio, and Francesco De Pellegrini. Demonstrating generalized virtual topologies in an openflow network. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 458–459, New York, NY, USA, 2011. ACM.

[54] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 265–274, New York, NY, USA, 2002. ACM.

[55] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, Srinivasan Seetharaman, David Underhill, Tatsuya Yabe, Kok-Kiong Yap, Yiannis Yiakoumis, Hongyi Zeng, Guido Appenzeller, Ramesh Johari, Nick McKeown, and Guru Parulkar. Carving research slices out of your production networks with OpenFlow. *SIGCOMM Comput. Commun. Rev.*, 40(1):129–130, January 2010.

[56] Rob Sherwood, Glen Gibb, Kok-kiong Yap, Guido Appenzeller, Martin Casado, Nick Mckeown, and Guru Parulkar. FlowVisor : A Network Virtualization Layer FlowVisor : A Network Virtualization Layer. *OpenFlow Switch*, page 15, 2009.

[57] H. Shimonishi and S. Ishii. Virtualized network infrastructure using OpenFlow. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pages 74–79, april 2010.

[58] Shunhong Song, Yuliang Lu, Weiwei Cheng, and Huan Yuan. Capability-centric attack model for network security analysis. In *Signal Processing Systems (ICSPS), 2010 2nd International Conference on*, volume 2, pages V2–372–V2–376, july 2010.

[59] N. Soudain, B.G. Raggad, and B. Zouari. A formal design of secure information systems by using a Formal Secure Data Flow Diagram (FSDFD). In *Risks and Security of Internet and Systems (CRiSIS), 2009 Fourth International Conference on*, pages 131–134, oct. 2009.

[60] Greg Stabler, Aaron Rosen, Sebastien Goasguen, and Kuang-Ching Wang. Elastic IP and security groups implementation using OpenFlow. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, VTDC '12, pages 53–60, New York, NY, USA, 2012. ACM.

[61] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. Network architecture for joint failure recovery and traffic engineering. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, pages 97–108, New York, NY, USA, 2011. ACM.

[62] Tcpdump. `http://www.tcpdump.org/`. Accessed on 02.04.2013.

[63] Chee-Wooi Ten, Chen-Ching Liu, and M. Govindarasu. Vulnerability Assessment of Cybersecurity for SCADA Systems Using Attack Trees. In *Power Engineering Society General Meeting, 2007. IEEE*, pages 1–8, june 2007.

[64] Gautam Thatte, Urbashi Mitra, and John Heidemann. Parametric methods for anomaly detection in aggregate traffic. *IEEE/ACM Trans. Netw.*, 19(2):512–525, April 2011.

[65] The Linux Foundation. netem. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`. Accessed on 02.04.2013.

[66] The Wireshark Foundation. Wireshark. `http://www.wireshark.org/`. Accessed on 02.04.2013.

[67] Michael Tunstall and Olivier Benoit. Efficient use of random delays in embedded software. In *Proceedings of the 1st IFIP TC6 /WG8.8 /WG11.2 international conference on Information security theory and practices: smart cards, mobile and ubiquitous computing systems*, WISTP'07, pages 27–38, Berlin, Heidelberg, 2007. Springer-Verlag.

[68] Nedeljko Vasić, Prateek Bhurat, Dejan Novaković, Marco Canini, Satyam Shekhar, and Dejan Kostić. Identifying and using energy-critical paths. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 18:1–18:12, New York, NY, USA, 2011. ACM.

[69] Linzhang Wang, E. Wong, and Dianxiang Xu. A Threat Model Driven Approach for Security Testing. In *Software Engineering for Secure Systems, 2007. SESS '07: ICSE Workshops 2007. Third International Workshop on*, page 10, may 2007.

[70] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[71] Wikipedia. Fault tree analysis. `http://en.wikipedia.org/wiki/Fault_tree_analysis`. Accessed on 02.04.2013.

[72] Ruoyu Wu, Weiguo Li, and He Huang. An Attack Modeling Based on Hierarchical Colored Petri Nets. In *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on*, pages 918–921, dec. 2008.

[73] Y. Yamasaki, Y. Miyamoto, J. Yamato, H. Goto, and H. Sone. Flexible Access Management System for Campus VLAN Based on OpenFlow. In *Applications and the Internet (SAINT), 2011 IEEE/IPSJ 11th International Symposium on*, pages 347–351, july 2011.

[74] Guang Yao, Jun Bi, and Peiyao Xiao. Source address validation solution with OpenFlow/NOX architecture. In *Network Protocols (ICNP), 2011 19th IEEE International Conference on*, pages 7–12, oct. 2011.

[75] Yang Yu, Chen Shanzhi, Li Xin, and Wang Yan. A framework of using OpenFlow to handle transient link failure. In *Transportation, Mechanical, and Electrical Engineering (TMEE), 2011 International Conference on*, pages 2050–2053, dec. 2011.

Appendix A

Diagram Legends

A.1 Data Flow Diagram Legend

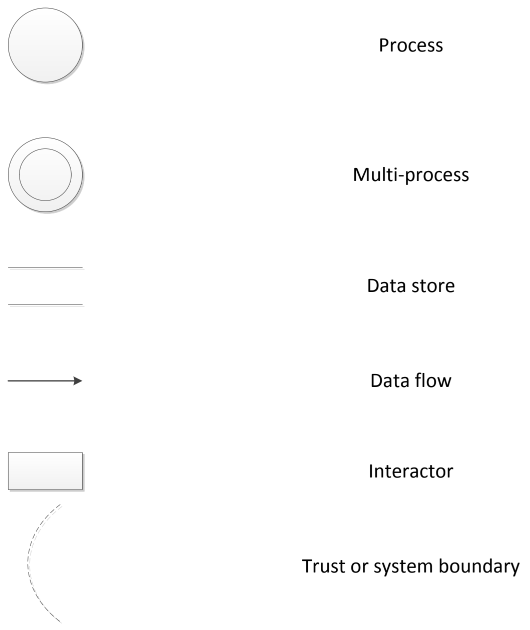


Figure A.1: Legend for data flow diagram

Element	Description
Process	Element that represents processes, including abstract processes
Data store	Element that models a store of data
Data flow	Element that models a flow of data between two other elements
Interactor	Element which represents external systems that interact with the system
Trust boundary	Element which separates elements at different level of trust

Table A.1: Description of data flow diagram elements

A.2 Attack Tree Legend

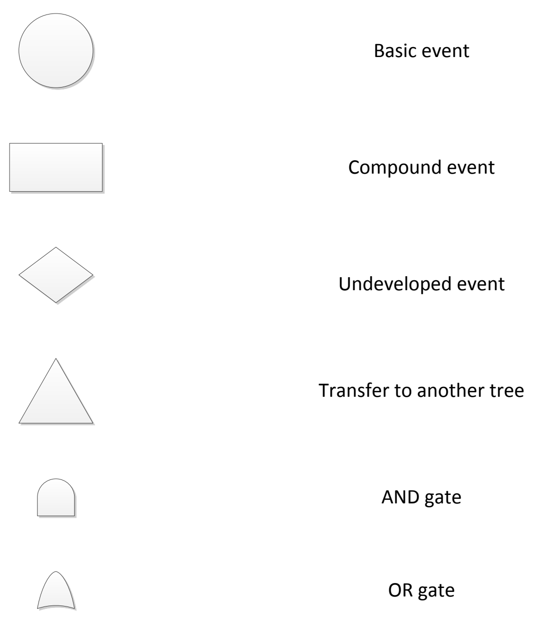


Figure A.2: Legend for attack trees

Element	Description
Basic element	A single action that can be readily performed
Compound element	A group of elements, to be further broken down
Undeveloped elements	A group of elements, without further description
Transfer to another tree	Attack tree is continued in another diagram
AND gate	All of the child elements must be executed
OR gate	One of the child elements must be executed

Table A.2: Description of attack tree diagram elements

Appendix B

Symbols Used

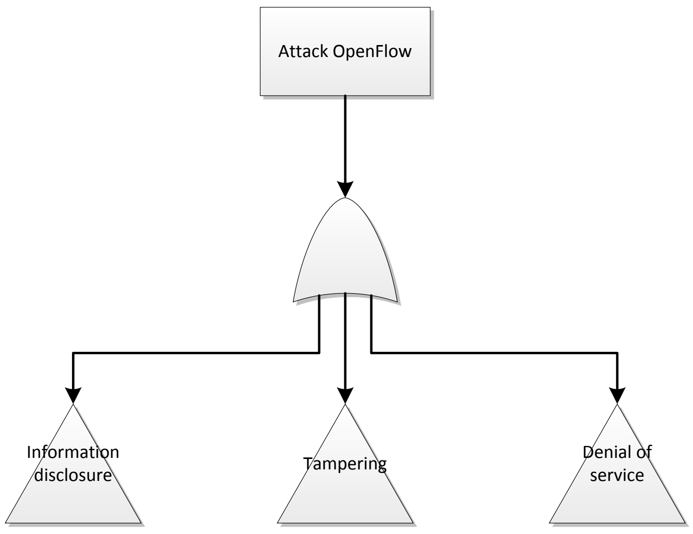
n_{bits}	is the size of a field in the flow table (in bits)
$n_{flow\,table}$	is the maximum number of entries a flow table
s_{pkt}	is the size of a packet used in an attack
s_{hdr}	is the size of the encapsulated header sent to the controller
$R_{flow\,rule}$	is the maximum number of flow rules that can be generated per second in a specific setup
R_{clnt}	is the throughput available between the client system performing an attack and the switch
R_{mgmt}	is the throughput available between the switch and the controller, including overhead for encryption
$t_{flow\,table}$	is the average dwelling time of a flow rule in the flow table (i.e. the flow rule timeout)
$t_{overflow}$	is the time required to effect an overflow
X_{switch}	is a random variable describing the packet processing time[1] for packets processed only in the switch (fast path)
$X_{controller}$	is a random variable describing the packet processing time for packets which must be forwarded to the controller
μ_{switch}	is the average (arithmetic mean) of X_{switch}
$\mu_{controller}$	is the average (arithmetic mean) of $X_{controller}$
σ_{switch}	is the standard deviation of X_{switch}
$\sigma_{controller}$	is the standard deviation of $X_{controller}$
$f_{X_{switch}}(x)$	is the probability distribution function of X_{switch}
$f_{X_{controller}}(x)$	is the probability distribution function of $X_{controller}$

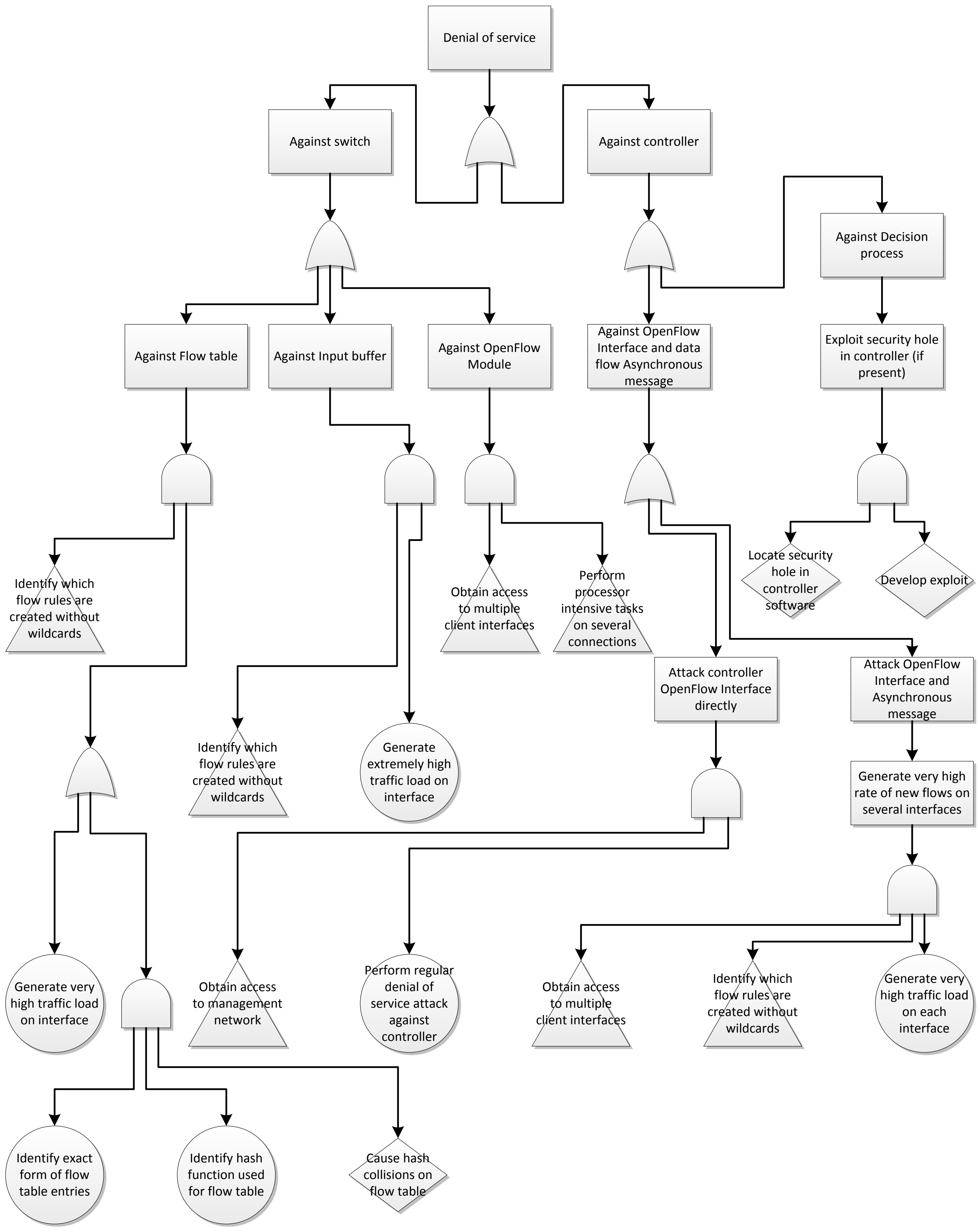
[1]Processing time is understood to be round trip time and transmission time

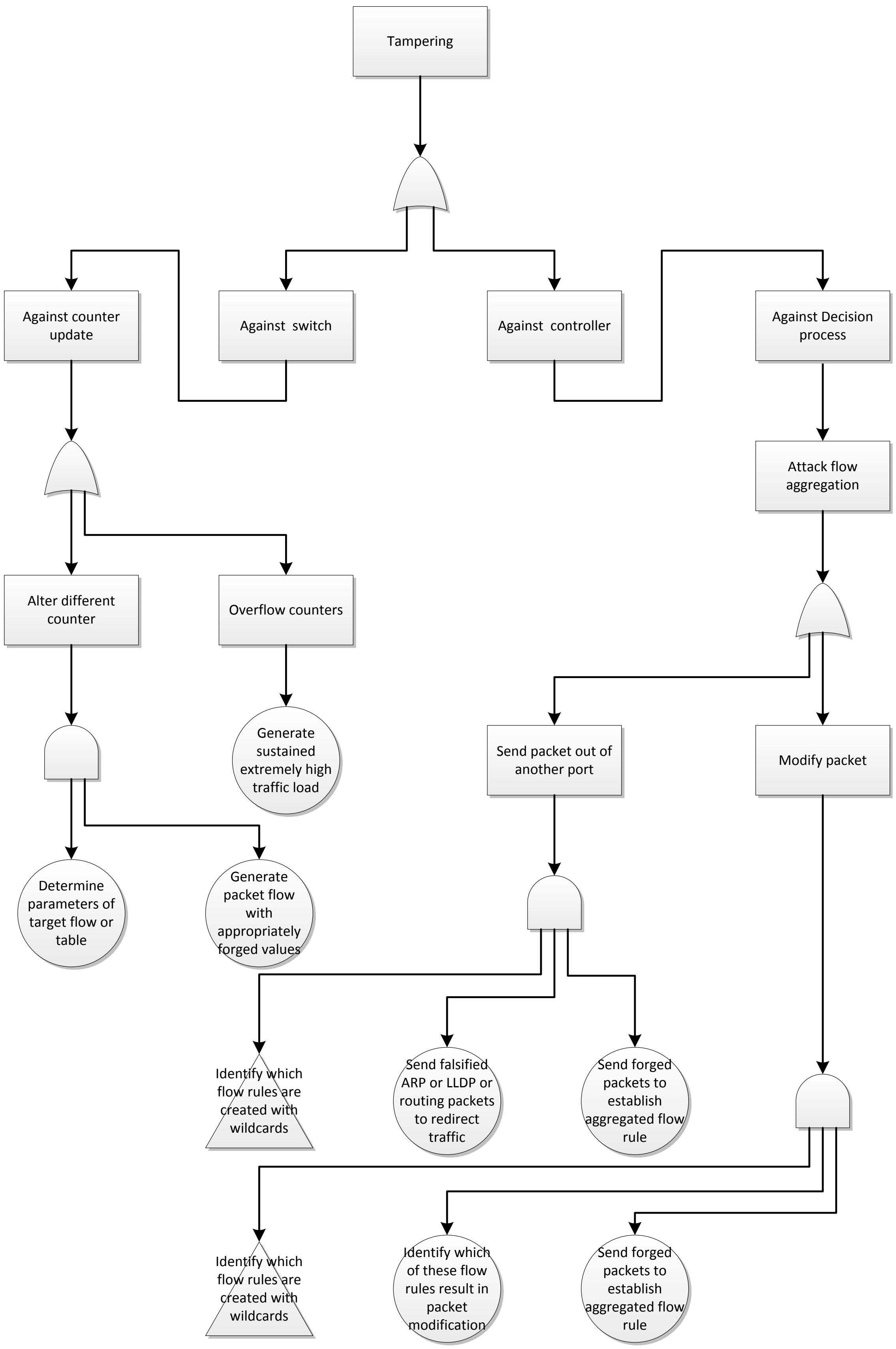
Appendix C

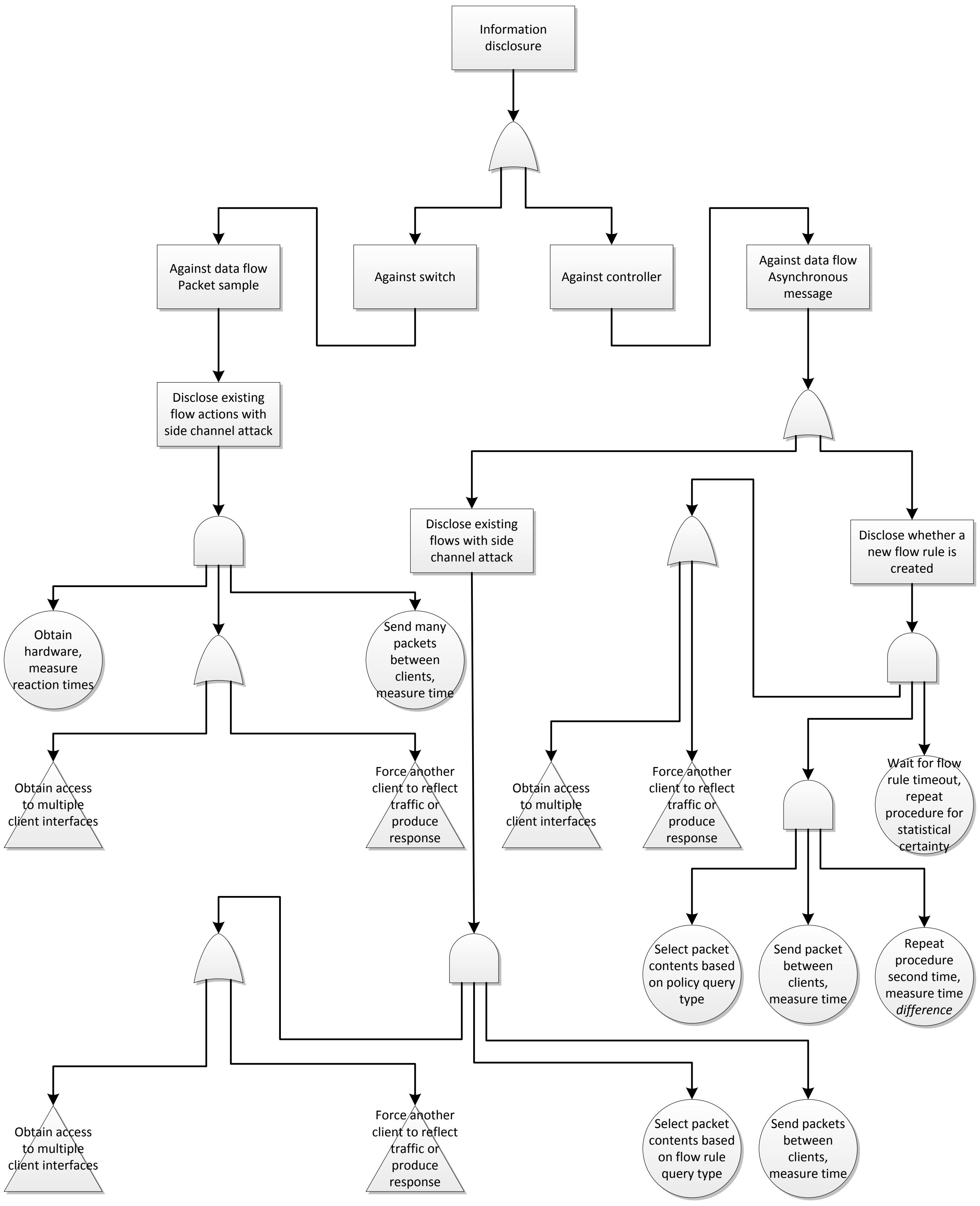
Attack Trees

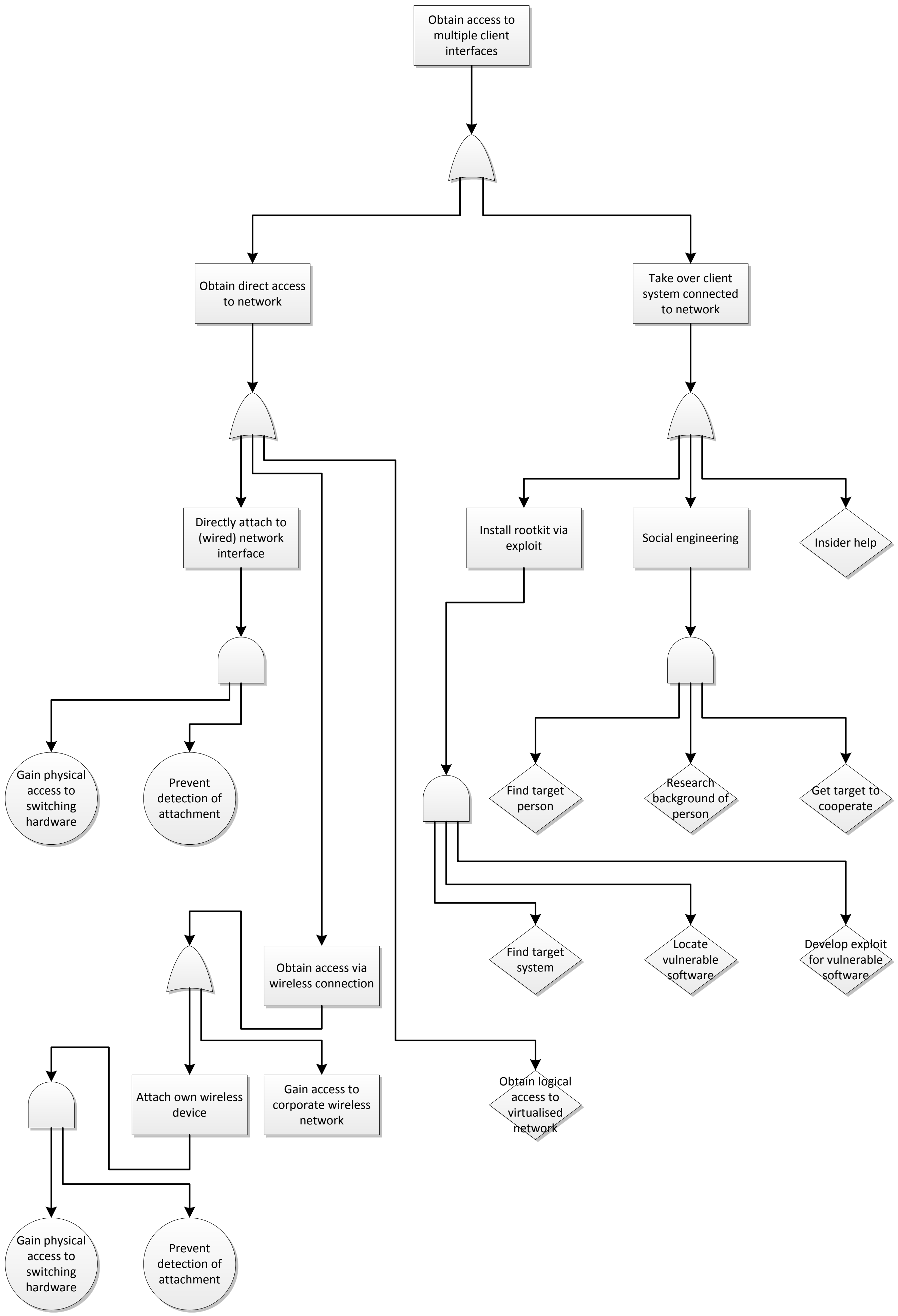
On the following pages, the full attack tree is reproduced.

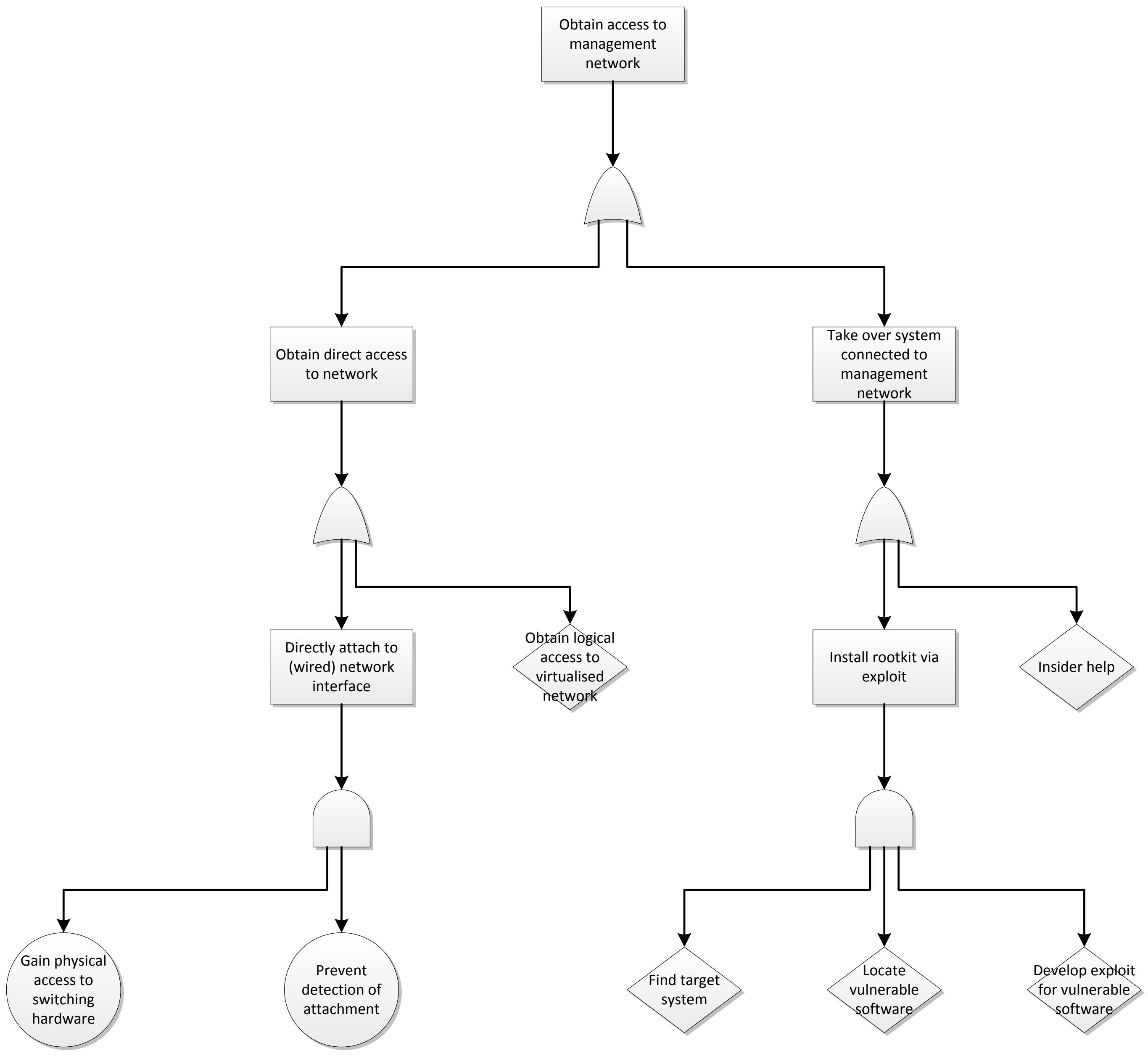

Attack OpenFlow
Information disclosure
Tampering
Denial of service

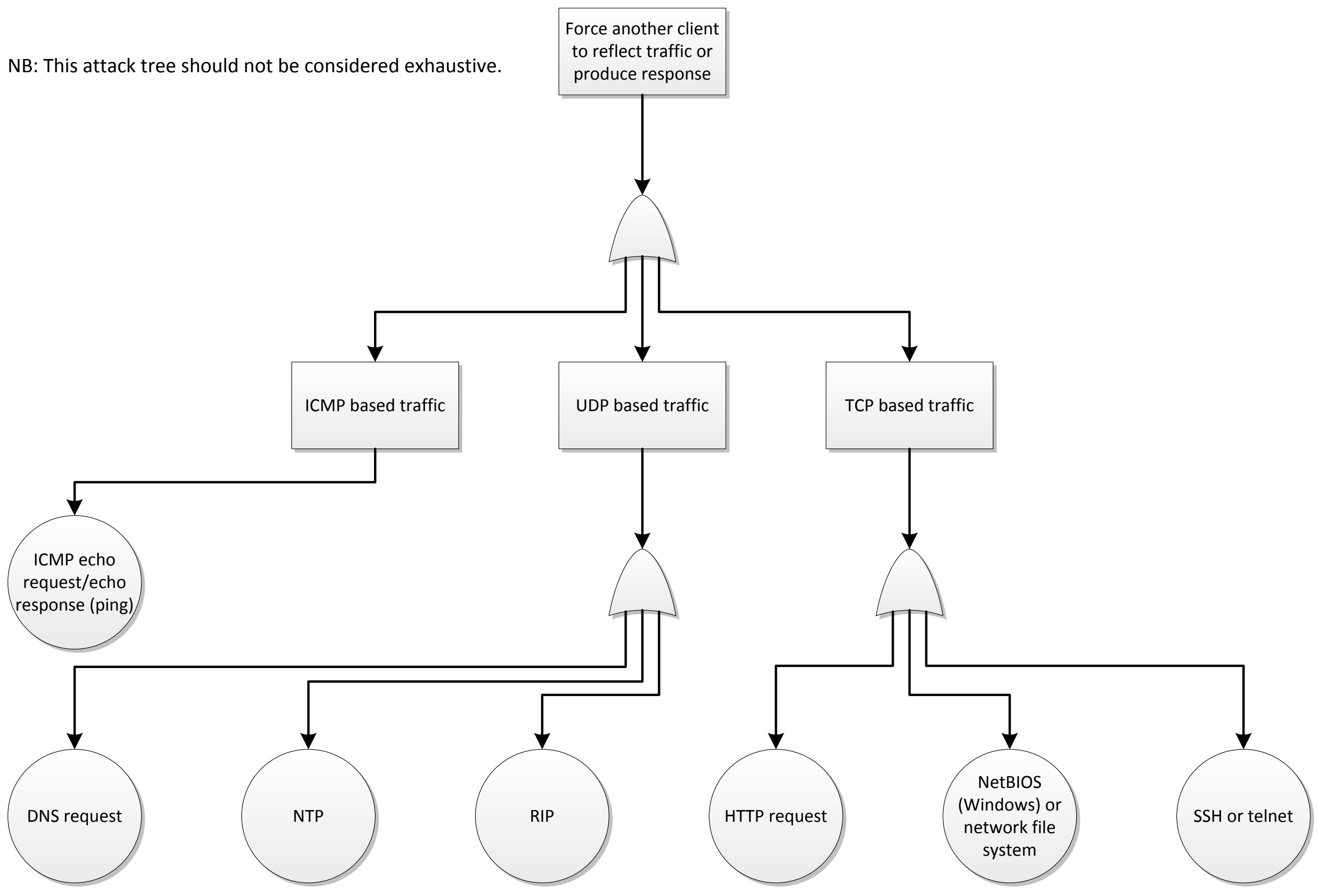

Denial of service
Against switch
Against controller
Against Decision process
Against Flow table
Against Input buffer
Against OpenFlow Module
Against OpenFlow Interface and data flow Asynchronous message
Exploit security hole in controller (if present)
Identify which flow rules are created without wildcards
Obtain access to multiple client interfaces
Perform processor intensive tasks on several connections
Locate security hole in controller software
Develop exploit
Attack controller OpenFlow Interface directly
Attack OpenFlow Interface and Asynchronous message
Identify which flow rules are created without wildcards
Generate extremely high traffic load on interface
Generate very high rate of new flows on several interfaces
Generate very high traffic load on interface
Obtain access to management network
Perform regular denial of service attack against controller
Obtain access to multiple client interfaces
Identify which flow rules are created without wildcards
Generate very high traffic load on each interface
Identify exact form of flow table entries
Identify hash function used for flow table
Cause hash collisions on flow table

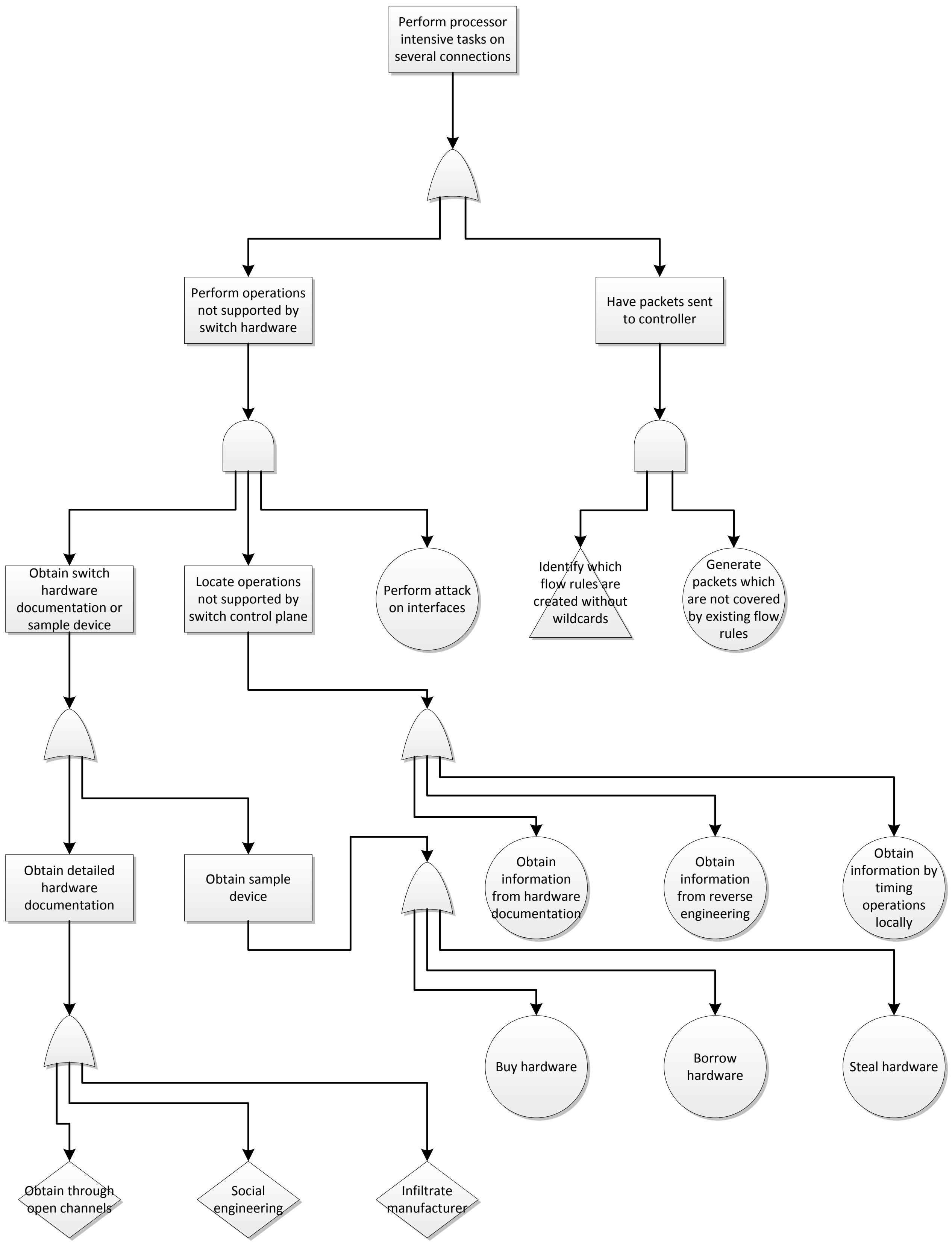

Tampering
Against counter update
Against switch
Against controller
Against Decision process
Attack flow aggregation
Alter different counter
Overflow counters
Generate sustained extremely high traffic load
Send packet out of another port
Modify packet
Determine parameters of target flow or table
Generate packet flow with appropriately forged values
Identify which flow rules are created with wildcards
Send falsified ARP or LLDP or routing packets to redirect traffic
Send forged packets to establish aggregated flow rule
Identify which flow rules are created with wildcards
Identify which of these flow rules result in packet modification
Send forged packets to establish aggregated flow rule

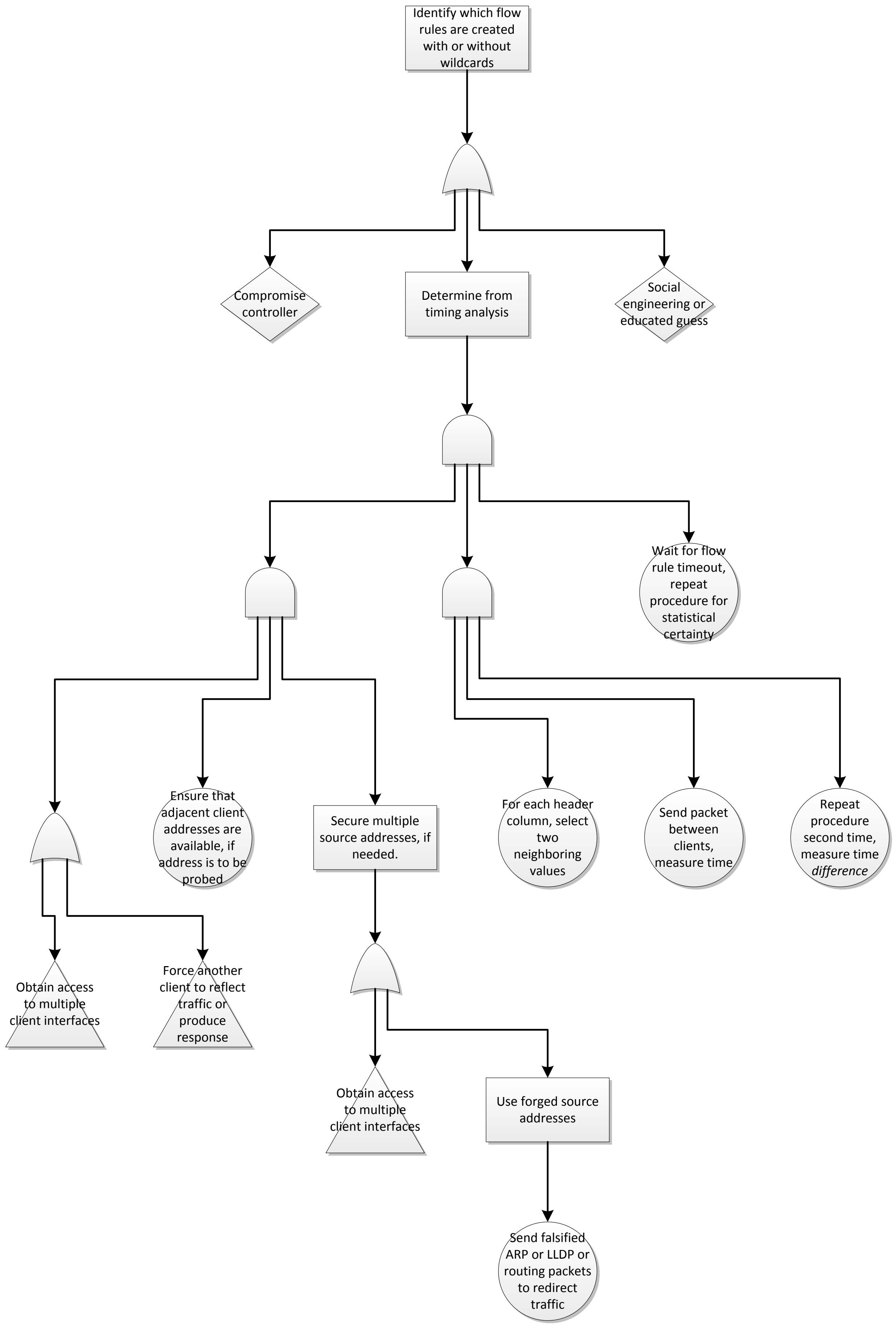

Information disclosure
Against data flow Packet sample
Against switch
Against controller
Against data flow Asynchronous message
Disclose existing flow actions with side channel attack
Disclose existing flows with side channel attack
Disclose whether a new flow rule is created
Obtain hardware, measure reaction times
Send many packets between clients, measure time
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Wait for flow rule timeout, repeat procedure for statistical certainty
Select packet contents based on policy query type
Send packet between clients, measure time
Repeat procedure second time, measure time *difference*
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Select packet contents based on flow rule query type
Send packets between clients, measure time


Obtain access to multiple client interfaces
Obtain direct access to network
Take over client system connected to network
Directly attach to (wired) network interface
Install rootkit via exploit
Social engineering
Insider help
Gain physical access to switching hardware
Prevent detection of attachment
Find target person
Research background of person
Get target to cooperate
Find target system
Locate vulnerable software
Develop exploit for vulnerable software
Obtain access via wireless connection
Attach own wireless device
Gain access to corporate wireless network
Obtain logical access to virtualised network
Gain physical access to switching hardware
Prevent detection of attachment


Obtain access to management network
Obtain direct access to network
Take over system connected to management network
Directly attach to (wired) network interface
Obtain logical access to virtualised network
Install rootkit via exploit
Insider help
Gain physical access to switching hardware
Prevent detection of attachment
Find target system
Locate vulnerable software
Develop exploit for vulnerable software


NB: This attack tree should not be considered exhaustive.
Force another client to reflect traffic or produce response
ICMP based traffic
UDP based traffic
TCP based traffic
ICMP echo request/echo response (ping)
DNS request
NTP
RIP
HTTP request
NetBIOS (Windows) or network file system
SSH or telnet


Perform processor intensive tasks on several connections
Perform operations not supported by switch hardware
Have packets sent to controller
Obtain switch hardware documentation or sample device
Locate operations not supported by switch control plane
Perform attack on interfaces
Identify which flow rules are created without wildcards
Generate packets which are not covered by existing flow rules
Obtain detailed hardware documentation
Obtain sample device
Obtain information from hardware documentation
Obtain information from reverse engineering
Obtain information by timing operations locally
Buy hardware
Borrow hardware
Steal hardware
Obtain through open channels
Social engineering
Infiltrate manufacturer


Identify which flow rules are created with or without wildcards
Compromise controller
Determine from timing analysis
Social engineering or educated guess
Wait for flow rule timeout, repeat procedure for statistical certainty
Ensure that adjacent client addresses are available, if address is to be probed
Secure multiple source addresses, if needed.
For each header column, select two neighboring values
Send packet between clients, measure time
Repeat procedure second time, measure time *difference*
Obtain access to multiple client interfaces
Force another client to reflect traffic or produce response
Obtain access to multiple client interfaces
Use forged source addresses
Send falsified ARP or LLDP or routing packets to redirect traffic

Appendix D

Code Listings

This appendix includes listings of all of the programs used in this thesis, as well as a brief description of their purpose. Most of these programs use either Scapy or Mininet, and therefore require root privileges on the system.

D.1 udp-multi

A short program which produces a sequence of UDP packets with ascending source and destination port numbers. The arguments for the program are *destination address*, *source address* and *port range* (maximum). The *source address* parameter can be use to spoof an alternative source address, if needed. For use with Mininet, it may be executed in an *XTerm* terminal started with the command `xterm h1` (for a host called *h1*). This program is used in D.6.

```
#!/usr/bin/env python

import sys
from os import popen
from scapy.all import sendp, IP, UDP, Ether

if len(sys.argv) != 4:
    print("Invalid␣arguments:␣" + sys.argv[0] + "␣<dst>␣<src>␣<port_count>")
    sys.exit(1)
else:
    dst_ip      = sys.argv[1]
    src_ip      = sys.argv[2]
    port_range  = int(sys.argv[3])

# Get correct interface name
interface       = popen('ifconfig␣|␣awk␣\'/eth0/␣{print␣$1}\'').read()

# Set of source and destination ports
port_set        = range(port_range)

# Construct set of UDP packets
packets = Ether()/IP(dst=dst_ip,src=src_ip)/UDP(dport=port_set,sport=port_set
    )

# Some feedback for the user
print('Created␣packet␣set:')
print(repr(packets))

# Send packets
# Note that Scapy will send the Cartesian product of all the setting domains,
     i.e. port_range^2 packets will be sent in total.
sendp(packets,iface=interface.rstrip())
```

Listing D.1: udp-multi.py

D.2 flowrule-overflow-test

This shell script makes it possible to empirically determine the number of flow rules that may be added to the flow table. The arguments for this program are *switch address* and *switch port*. These specify where the switch can be found, as used with the *ovs-ofctl* command.

```bash
#!/usr/bin/env bash

# Usage: flowrule-overflow-test <Switch address> <Switch port>
SWITCH_ADDR=$1
SWITCH_PORT=$2

# Install flow rule with specified TCP source and destination ports
install_flow_rule() {
    ovs-ofctl add-flow tcp:$SWITCH_ADDR:$SWITCH_PORT tcp,tp_src=$1,tp_dst=$2,idle_timeout=0,actions=output:1
}

# Remove all existing flow rules
clear_flow_rules() {
    ovs-ofctl del-flows tcp:$SWITCH_ADDR:$SWITCH_PORT
}

# Query the number of flow rules installed
count_flow_rules() {
    ovs-ofctl dump-flows tcp:$SWITCH_ADDR:$SWITCH_PORT | awk '{N++} END {print N}'
}

# Ensure that OpenFlow switch is running
CONNECTIVITY_CHECK=`ovs-ofctl show tcp:$SWITCH_ADDR:$SWITCH_PORT | awk '/OFPT_FEAT/ {print $1}'`

# Exit if the switch is not running, otherwise clear switch flow table
if [[ $CONNECTIVITY_CHECK != "OFPT_FEATURES_REPLY" ]]; then
    echo "Unable␣to␣contact␣switch."
    exit;
else
    clear_flow_rules
fi

# Initialise variables to contain counters
FLOW_RULE_COUNT=0
FLOW_RULE_COUNT_OLD=-1
ITERATOR=0

# Keep adding flow rules until total number of flow rules stops increasing
until [[ $FLOW_RULE_COUNT -le $FLOW_RULE_COUNT_OLD ]]; do
    FLOW_RULE_COUNT_OLD=$FLOW_RULE_COUNT
    install_flow_rule $[ $ITERATOR / 65534 + 1] $[ $ITERATOR % 65534 + 1]
    FLOW_RULE_COUNT=`count_flow_rules`
    if [[ $[ $FLOW_RULE_COUNT % 1000 ] -eq 0 ]]; then
        echo "$FLOW_RULE_COUNT␣flow␣rules␣created␣so␣far..."
    fi
    let "ITERATOR++"
done

# Cleanup, then print results
clear_flow_rules
echo "Created␣$FLOW_RULE_COUNT␣flow␣rules␣in␣total"
```

Listing D.2: flowrule-overflow-test.sh

D.3 mininet-init

This program demonstrates the use of the Mininet API to establish a Mininet network, with the topography described in 5.3.1. The arguments for this program are *bandwidth* (simulated link bandwidth in Mbps), *delay* (in ms) and *loss* (of packets in percent). This can be used to establish a Mininet setup without making use of the interactive CLI. Here, the use of link parameters is demonstrated, and the *iperf* test is performed to measure available bandwidth.

```python
#!/usr/bin/env python
import sys
from mininet.net import Mininet
from mininet.topo import SingleSwitchTopo
from mininet.link import TCLink

if len(sys.argv) != 4:
    print("Usage: " + sys.argv[0] + " <bandwidth [Mbps]> <delay [ms]> <loss [%]>")
    sys.exit(0)
else:
    bandwidth           = int(sys.argv[1])
    delay               = '{0}ms'.format(int(sys.argv[2]))
    loss                = int(sys.argv[3])

# Configure link parameters
link_params = {
    'bw'                : bandwidth,
    'delay'             : delay,
    'loss'              : loss,
    'max_queue_size'    : 10000,
    'use_htb'           : True
}

# Establish net topology (3 hosts connected to one switch)
topology    = SingleSwitchTopo(k=3,lopts=link_params)

# Establish network with the specified topology
net         = Mininet(topo=topology,link=TCLink)

# Initialise network
net.start()

# Get node objects
h1          = net.hosts[0]
h2          = net.hosts[1]
h3          = net.hosts[2]
s1          = net.switches[0]
c1          = net.controllers[0]

# Get link objects from switch
l1          = s1.connectionsTo(h1)[0][0].link
l2          = s1.connectionsTo(h2)[0][0].link
l3          = s1.connectionsTo(h3)[0][0].link

# Perform action - test network performance
print(net.iperf(hosts=[h1,h2]))

# Shut network down
net.stop()
```

Listing D.3: mininet-init.py

D.4 mininet-custom-topo

D.4.1 Usage

This program extends the Mininet API to add a custom topology, allowing different performance limitations to be imposed on individual links. This program should be started with the name of a configuration file in the JSON format.

Parameter	Description
`data`	Parameters for data path network connections
`control`	Parameters for control path network connections
`ctrl_exec`	Command to execute, including path but not parameters
`ctrl_logf`	File to log to
`ctrl_logl`	Log level, in all caps
`ctrl_args`	Other modules to load, with arguments
`ports_max`	See Section D.5
`id_wait`	See Section D.7

Table D.1: Parameters for mininet-custom-topo

For the contents of the *data* and a *control* section, see the link paramters for the *TCIntf* class[1]. The program creates a subclass of the *topology* class and specifies that the switch-controller link be placed in a separate namespace just as the host-switch links are. The *RemoteController* class is used to execute the POX controller[2] with specified modules. It also sets performance limits on the control link and data link (these can be different). After the network is established, *iperf* is used to test connectivity. Partially adapted from a sample in the Mininet documentation[3].

```
{
    "control"   : {
        "bw"                : 100,
        "delay"             : "10ms",
        "jitter"            : null,
        "loss"              : 0,
        "max_queue_size"    : null,
        "speedup"           : 0,
        "use_htb"           : true
    },
    "data"      : {
        "bw"                : 100,
        "delay"             : "1ms",
        "jitter"            : null,
        "loss"              : 0,
        "max_queue_size"    : null,
        "speedup"           : 0,
        "use_htb"           : true
    },
    "ctrl_exec" : "/home/mininet/src/pox/pox.py",
    "ctrl_logf" : "/home/mininet/src/output/c0.log",
    "ctrl_logl" : "DEBUG",
    "ctrl_args" : "forwarding.l2_learning",
    "ports_max" : 800,
    "id_wait"   : 30
}
```

Listing D.4: Example of JSON parameters file

[1]See http://mininet.github.com/api/classmininet_1_1link_1_1TCIntf.html
[2]See http://www.noxrepo.org/pox/about-pox/
[3]See https://github.com/mininet/mininet/wiki/Introduction-to-Mininet

D.4.2 Listing

```python
#!/usr/bin/env python
import sys
import json
from mininet.net import MininetWithControlNet
from mininet.node import CPULimitedHost
from mininet.node import UserSwitch
from mininet.node import RemoteController
from mininet.topo import Topo
from mininet.link import TCIntf
from mininet.link import TCLink

# Process command line
if len(sys.argv) != 2:
    print('Usage:␣{0}␣<params_JSON>'.format(sys.argv[0]))
    sys.exit(0)
else:
    # Open and parse configuration file (in JSON format)
    cfile   = open(sys.argv[1])
    cfg     = json.load(cfile)
    cfile.close()

    # Get parameters
    link_params_data    = cfg['data']
    link_params_control = cfg['control']
    ctrl_exec           = cfg['ctrl_exec']
    ctrl_logf           = cfg['ctrl_logf']
    ctrl_logl           = cfg['ctrl_logl']
    ctrl_args           = cfg['ctrl_args']

# For formatting a list with newlines and filler characters
def join_elements(elements,function):
    'Join␣elements␣with␣newline␣and␣alignment'
    def append(a,b): return a + '\n' + ('.' * 17) + b
    return reduce(append,map(function,elements))

# This is so we can get the link elements from a node
def get_links(node):
    'Return␣all␣link␣elements␣for␣a␣given␣node'
    def link(x): return x.link
    return map(link,node.intfList())

# Create class derived from base topology class
class CustomTopo(Topo):
    'Custom␣class␣to␣implement␣performance␣constraints␣on␣connections'
    def __init__(self, n=3, **opts):
        # Call base class constructor
        Topo.__init__(self, **opts)

        # Create new switch s1
        s1          = self.addSwitch('s1')

        # Create hosts and connect to switch
        for i in range(n):
            name    = 'h{0}'.format(i + 1)
            h       = self.addHost(name)
            self.addLink(s1, h, **opts['lopts'])

# Network parameters, specified here for the sake of clarity
net_params  = {
    'topo'              : CustomTopo(n=3,lopts=link_params_data),
    'cleanup'           : True,
    'inNamespace'       : True,
    'listenPort'        : 6634,
    'host'              : CPULimitedHost,
    'switch'            : UserSwitch,
    'controller'        : RemoteController,
    'link'              : TCLink,
    'intf'              : TCIntf
}

# Starting time
start_time = int(time.time())

# Establish network with the specified topology
net         = MininetWithControlNet(**net_params)

# Initialise network
net.start()

# Get node objects
h1          = net.hosts[0]
h2          = net.hosts[1]
s1          = net.switches[0]
c0          = net.controllers[0]

# Get interface objects from switch
i1          = s1.connectionsTo(h1)[0][0]
i2          = h2.connectionsTo(s1)[0][0]
i0          = s1.connectionsTo(c0)[0][0]

# Execute controller on controller node
ctrl_addr = 'openflow.of_01␣--address={0}␣--port={1}'.format(c0.IP(),c0.port)
ctrl_logp = 'log␣--file={0}␣log.level␣--{1}'.format(ctrl_logf,ctrl_logl)
ctrl_cmdl = '{0}␣{1}␣{2}␣{3}'.format(ctrl_exec,ctrl_args,ctrl_logp,ctrl_addr)
ctrl_out = open('output/of-out-{0}'.format(start_time),'w')
ctrl_err = open('output/of-err-{0}'.format(start_time),'w')
ctrl_popn = c0.popen(ctrl_cmdl,stdout=ctrl_out,stderr=ctrl_err,shell=True)

# Set performance specifications for controller link
i0.config(**link_params_control)

# Print out network metadata
print('Created␣network:')
print('Hosts...........{0}'.format(join_elements(net.hosts,repr)))
print('Switches.........{0}'.format(join_elements(net.switches,repr)))
print('Controllers......{0}'.format(join_elements(net.controllers,repr)))
print('Switch␣ports.....{0}'.format(join_elements(get_links(s1),str)))

# Perform action - test network performance
print(net.iperf(hosts=[h1,h2]))

# Terminate controller. This must be done manually, as we are using POX.
ctrl_popn.terminate()
ctrl_out.close()
ctrl_err.close()

# Shut network down
net.stop()
```

Listing D.5: mininet-custom-topo.py

D.5 attack-demo-dos

This program is closely based on D.5, but actually uses D.1 to perform an attack from host 1 to host 2. This program is supplied a JSON configuration file, as in Section D.4. However, it should also contain an integer *ports_max* (this is actually the square root of the number of packets that will be sent). It spawns an instance of *tcpdump* on the switch to intercept the traffic on the control channel. This packet trace can then be inspected in *Wireshark*. It spawns a second instance on the receiving ("victim") host, in order to record the number of received packets.

```
#!/usr/bin/env python
import sys
import os
import time
import json
from mininet.net import MininetWithControlNet
from mininet.node import CPULimitedHost
from mininet.node import UserSwitch
from mininet.node import RemoteController
from mininet.topo import Topo
from mininet.link import TCIntf
from mininet.link import TCLink
from mininet.log import setLogLevel

# Process command line
if len(sys.argv) != 2:
    print("Usage:␣" + sys.argv[0] + "␣<params_JSON>")
    sys.exit(0)
else:
    # Open and parse configuration file (in JSON format)
    cfile   = open(sys.argv[1])
    cfg     = json.load(cfile)
    cfile.close()

    # Get parameters
    link_params_data    = cfg['data']
    link_params_control = cfg['control']
    ctrl_exec           = cfg['ctrl_exec']
    ctrl_logf           = cfg['ctrl_logf']
    ctrl_logl           = cfg['ctrl_logl']
    ctrl_args           = cfg['ctrl_args']
    port_count          = cfg['ports_max']

# For formatting a list with newlines and filler characters
def join_elements(elements,function):
    'Join␣elements␣with␣newline␣and␣alignment'
    def append(a,b): return a + '\n' + ('.' * 17) + b
    return reduce(append,map(function,elements))

# This is so we can get the link elements from a node
def get_links(node):
    'Return␣all␣link␣elements␣for␣a␣given␣node'
    def link(x): return x.link
    return map(link,node.intfList())

# Create class derived from base topology class
class CustomTopo(Topo):
    'Custom␣class␣to␣implement␣performance␣constraints␣on␣connections'
    def __init__(self, n=3, **opts):
        # Call base class constructor
        Topo.__init__(self, **opts)

        # Create new switch s1
        s1          = self.addSwitch('s1')

        # Create hosts and connect to switch
        for i in range(n):
            name    = 'h{0}'.format(i + 1)
            h       = self.addHost(name)
            self.addLink(s1, h, **opts['lopts'])

# Network parameters, specified here for the sake of clarity
net_params  = {
    'topo'              : CustomTopo(n=3,lopts=link_params_data),
    'cleanup'           : True,
    'inNamespace'       : True,
    'listenPort'        : 6634,
    'host'              : CPULimitedHost,
    'switch'            : UserSwitch,
    'controller'        : RemoteController,
    'link'              : TCLink,
    'intf'              : TCIntf
}

# Starting time
start_time = int(time.time())

# Set level of logging
setLogLevel('info')

# Establish network with the specified topology
net         = MininetWithControlNet(**net_params)

# Initialise network
net.start()

# Get node objects
h1          = net.hosts[0]
h2          = net.hosts[1]
s1          = net.switches[0]
c0          = net.controllers[0]

# Get interface objects from switch
i1          = s1.connectionsTo(h1)[0][0]
i2          = h2.connectionsTo(s1)[0][0]
i0          = s1.connectionsTo(c0)[0][0]

# Execute controller on controller node
ctrl_addr = 'openflow.of_01␣--address={0}␣--port={1}'.format(c0.IP(),c0.port)
ctrl_logp = 'log␣--file={0}␣log.level␣--{1}'.format(ctrl_logf,ctrl_logl)
ctrl_cmdl = '{0}␣{1}␣{2}␣{3}'.format(ctrl_exec,ctrl_args,ctrl_logp,ctrl_addr)
ctrl_out = open('output/of-out-{0}'.format(start_time),'w')
ctrl_err = open('output/of-err-{0}'.format(start_time),'w')
ctrl_popn = c0.popen(ctrl_cmdl,stdout=ctrl_out,stderr=ctrl_err,shell=True)

# Set performance specifications for controller link
i0.config(**link_params_control)

# Print out network metadata
print('Created␣network:')
print('Hosts...........{0}'.format(join_elements(net.hosts,repr)))
print('Switches........{0}'.format(join_elements(net.switches,repr)))
print('Controllers.....{0}'.format(join_elements(net.controllers,repr)))
print('Switch␣ports....{0}'.format(join_elements(get_links(s1),str)))

# Execute tcpdump client on switch to capture packets
interface_name      = str(i0)
dump_name           = 'output/of-packetdump-{0}'.format(start_time)
```

```
pcap_filter         = ’tcp␣and␣port␣6633’
s1.cmd(’tcpdump␣-i␣{0}␣-w␣{1}␣-U␣{2}␣&’.format(interface_name,dump_name,
    pcap_filter))

# Execute tcpdump client on host 2 to count number of received packets
interface_name      = str(i2)
output_name         = ’output/of-client-recv-{0}’.format(start_time)
pcap_filter         = ’dst␣{0}␣and␣udp’.format(i2.IP())
h2.cmd(’tcpdump␣-i␣{0}␣-w␣/dev/null␣-s␣256␣-q␣{1}␣&>␣{2}␣&’.format(
    interface_name,pcap_filter,output_name))

# Execute attack client on host
h1.cmd(’./udp-multi.py␣{0}␣{1}␣{2}’.format(h2.IP(),h1.IP(),port_count))

# Send interrupt to stop tcpdump
time.sleep(1)
s1.cmd(’yes␣|␣killall␣-int␣tcpdump’)
h2.sendInt()
time.sleep(1)

# Write protocol statistics to file
f = open(’output/of-protostat-{0}’.format(start_time),’w’)
f.write(s1.dpctl(’show-protostat’))
f.close()

# Terminate controller. This must be done manually, as we are using POX.
ctrl_popn.terminate()
ctrl_out.close()
ctrl_err.close()

# Shut network down
net.stop()
```

Listing D.6: attack-demo-dos.py

D.6 dos-timeout-probe

This shell script executes attack-demo-dos several times, with an increasing soft timeout, which is passed to the POX module by means of the environment variable *TIMEOUT*[4]. This is to attempt to correlate the value of the timeout and the success of the attack. The results are stored in a CSV file, specified in the source code. The arguments are *start*, *end* and *step*. The first two values specify the starting and finishing timeout values, respectively. The latter value specifies the increment after each run.

```bash
#!/usr/bin/env bash

# This script is intended to probe the results of a denial of service
# attack with different soft timeout values. The script attempts the attack
# using values between START and END, with an increment of STEP each
    iteration.
#
# Usage: dos_timeout_probe <Start timeout> <End timeout> <Timeout step>
START=$1
END=$2
STEP=$3

# Command to execute
CMD="python␣attack-demo-dos.py␣params.json"

# File to store results to
OUTPUT_FILE="output/attack-stats.csv"

# Initialise file
echo "\"Timeout [s]\",\"Timestamp [s]\",\"Execution time [s]\",\"Packet In\"
    ,\"Packet Out\",\"Flow Mod\",\"Error\",\"Flows added\",\"Flows deleted\"
    ,\"All tables full\",\"Packets received at client\"" > $OUTPUT_FILE

# Function to get latest timestamp
get_timestamp() {
    ls -l output/of-packetdump-* | awk '{print $9}' | sort | tail -n 1 | cut
        -d- -f3
}

# Function to execute attack
execute_attack() {
    SOFT_TIMEOUT=$1
    echo "Executing:␣[sudo␣SOFT_TIMEOUT=${SOFT_TIMEOUT}␣$CMD]"
    sudo SOFT_TIMEOUT=${SOFT_TIMEOUT} $CMD
}

# Clear up junk
cleanup_files() {
    TIMESTAMP=$1

    # These files are generated by attack-demo-dos.py
    yes | sudo rm output/of-protostat-$TIMESTAMP
    yes | sudo rm output/of-client-recv-$TIMESTAMP
    yes | sudo rm output/of-out-$TIMESTAMP
    yes | sudo rm output/of-err-$TIMESTAMP
    yes | sudo rm output/of-packetdump-$TIMESTAMP
    yes | sudo rm output/c0.log

    # Ensure that no processes are left over
    ps aux | awk '/ofprotocol|controller|ofdatapath|ovs-controller|ovsdb-
        server|ovs-vswitchd|python|tcpdump/ {print $2}' | sort | uniq | sudo
        xargs kill

```

[4]The value is in seconds

```
    # Remove sockets created by OpenvSwitch, if necessary
    sudo rm -rf /tmp/vconn-unix.* > /dev/null
}

# Function to compile results
# The results will be outputted as a CSV file (comma delimiter, linefeed record separator, no quoting)
process_results() {
    VALUE_IDX=$1
    TIME=$2
    TIMESTAMP=`get_timestamp`
    STATS_FILE="output/of-protostat-${TIMESTAMP}"
    CLIENT_FILE="output/of-client-recv-${TIMESTAMP}"
    VALUE_PKT_IN=`awk '/packet in/ && FLAG {print $1} /packet in/ && !FLAG {FLAG=1}' $STATS_FILE`
    VALUE_PKT_OUT=`awk '/packet out/ && !FLAG {print $1; FLAG=1}' $STATS_FILE`
    VALUE_FLOW_MOD=`awk '/flow mod/ && !FLAG {print $4; FLAG=1}' $STATS_FILE`
    VALUE_ERR=`awk '/errors/ && FLAG {print $3} /errors/ && !FLAG {FLAG=1}' $STATS_FILE`
    VALUE_FLOW_ADD=`awk '/add/ && !FLAG {print $2; FLAG=1}' $STATS_FILE`
    VALUE_FLOW_DEL=`awk '/delete/ && !FLAG {print $1; FLAG=1}' $STATS_FILE`
    VALUE_TABL_FULL=`awk '/tables full/ && FLAG {print $5} /tables full/ && !FLAG {FLAG=1}' $STATS_FILE`
    VALUE_PKT_RECV=`awk '/captured/ {print $1}' $CLIENT_FILE`
    OUTPUT_LINE="$VALUE_IDX,$TIMESTAMP,$TIME,$VALUE_PKT_IN,$VALUE_PKT_OUT,$VALUE_FLOW_MOD,$VALUE_ERR,$VALUE_FLOW_ADD,$VALUE_FLOW_DEL,$VALUE_TABL_FULL,$VALUE_PKT_RECV"
    echo "Results␣of␣attack␣probe␣(Timeout␣value:␣[$VALUE_IDX]␣-␣Timestamp:␣[$TIMESTAMP]):␣[$OUTPUT_LINE]"
    echo $OUTPUT_LINE >> $OUTPUT_FILE
    cleanup_files $TIMESTAMP
}

# Initialise counter
VALUE=$START

# Main loop
while [[ $VALUE -le $END ]]; do
    echo "Attempting␣to␣perform␣attack␣with␣timeout:␣[$VALUE]"
    START_TIME=`date +%s`
    execute_attack $VALUE
    END_TIME=`date +%s`
    let "TIME␣=␣END_TIME␣-␣START_TIME"
    process_results $VALUE $TIME
    let "VALUE␣=␣VALUE␣+␣STEP"
done
```

Listing D.7: dos-timeout-probe.sh

D.7 attack-demo-id

This program is a further development on the one described in Section D.5. This program is supplied a JSON configuration file, as in Section D.4 and Section D.5. However, it should also contain an integer *id_wait*, which is the number of seconds required for a flow rule to time out. The program creates a network based on the topology in Figure 5.2 on page 50. On the right side, a server is established[5]. A connection is established from the attacking system to the server; this allows the controller to learn about the attacking client. Then the flow rule is allowed to expire. Two measurements are made on test system, then a second period is waited for the flow rules to be removed due to timeout. Then, client connections on either of the test systems neighbours are established[6]. A second set of measurements is then performed.

```python
#!/usr/bin/env python
import sys
import os
import time
import json
import re
from subprocess import PIPE
from mininet.net import MininetWithControlNet
from mininet.node import CPULimitedHost
from mininet.node import UserSwitch
from mininet.node import RemoteController
from mininet.topo import Topo
from mininet.link import TCIntf
from mininet.link import TCLink
from mininet.log import setLogLevel

# Process command line
if len(sys.argv) != 2:
    print("Usage: " + sys.argv[0] + " <params_JSON>")
    sys.exit(0)
else:
    # Open and parse configuration file (in JSON format)
    cfile   = open(sys.argv[1])
    cfg     = json.load(cfile)
    cfile.close()

    # Get parameters
    link_params_data    = cfg['data']
    link_params_control = cfg['control']
    ctrl_exec           = cfg['ctrl_exec']
    ctrl_logf           = cfg['ctrl_logf']
    ctrl_logl           = cfg['ctrl_logl']
    ctrl_args           = cfg['ctrl_args']
    port_count          = cfg['ports_max']
    id_wait             = cfg['id_wait']

# For formatting a list with newlines and filler characters
def join_elements(elements,function):
    'Join elements with newline and alignment'
    def append(a,b): return a + '\n' + ('.' * 17) + b
    return reduce(append,map(function,elements))

# This is so we can get the link elements from a node
def get_links(node):
    'Return all link elements for a given node'
    def link(x): return x.link
    return map(link,node.intfList())

# This probes a host and returns the time required for a TCP SYN request
```

[5] Actually only *netcat* listening on port 80
[6] Again using *netcat*

```
# to be answered.
def probe_time(host, interface, target):
    'Return␣response␣time␣for␣TCP␣SYN␣packet'
    args_ip     = '{0}␣{1}'.format(target.IP(),interface.IP())
    args_mac    = '{0}␣{1}'.format(target.MAC(),interface.MAC())
    args        = '{0}␣{1}␣{2}␣80'.format(args_ip,args_mac,str(interface))
    query       = 'python␣tcp-time.py␣{0}'.format(args)
    print('Host:␣[{0}]␣-␣Command:␣[{1}]'.format(str(host),query))

    # Run the probe.
    output = host.cmd(query)

    # This regular expression is used to extract the time from the
    # process output.
    regex = 'Time:␣\\[([0-9\\.]+)\\]'
    return float(re.search(regex, output).group(1))

# Create class derived from base topology class
class CustomTopo(Topo):
    'Custom␣class␣to␣implement␣performance␣constraints␣on␣connections'
    def __init__(self, n=3, **opts):
        # Call base class constructor
        Topo.__init__(self, **opts)

        # Create new switch s1
        s1          = self.addSwitch('s1')

        # Create new switch s2
        s2          = self.addSwitch('s2')

        # Link the switches together
        self.addLink(s1, s2, **opts['lopts'])

        # Create hosts and connect to switch 1
        for i in range(n):
            name    = 'h1-{0}'.format(i + 1)
            h       = self.addHost(name)
            self.addLink(s1, h, **opts['lopts'])

        # Create second set of hosts and connect to switch 2
        for i in range(n):
            name    = 'h2-{0}'.format(i + 1)
            h       = self.addHost(name)
            self.addLink(s2, h, **opts['lopts'])

# Network parameters, specified here for the sake of clarity
net_params  = {
    'topo'              : CustomTopo(n=3,lopts=link_params_data),
    'cleanup'           : True,
    'inNamespace'       : True,
    'listenPort'        : 6634,
    'host'              : CPULimitedHost,
    'switch'            : UserSwitch,
    'controller'        : RemoteController,
    'link'              : TCLink,
    'intf'              : TCIntf
}

# Starting time
start_time = int(time.time())

# Set level of logging
setLogLevel('info')
```

```

# Establish network with the specified topology
net          = MininetWithControlNet(**net_params)

# Initialise network
net.start()

# Get node objects
h1_1         = net.hosts[0]
h1_2         = net.hosts[1]
h1_3         = net.hosts[2]
h2_1         = net.hosts[3]
h2_2         = net.hosts[4]
h2_3         = net.hosts[5]
s1           = net.switches[0]
s2           = net.switches[1]
c0           = net.controllers[0]

# Get interface objects from switch
i1_1         = s1.connectionsTo(h1_1)[0][0]
i1_2         = h1_2.connectionsTo(s1)[0][0]
i1_0         = s1.connectionsTo(c0)[0][0]
i2_0         = s2.connectionsTo(c0)[0][0]
i2_1         = h2_1.connectionsTo(s2)[0][0]

# Execute controller on controller node
ctrl_addr = 'openflow.of_01␣--address={0}␣--port={1}'.format(c0.IP(),c0.port)
ctrl_logp = 'log␣--file={0}␣log.level␣--{1}'.format(ctrl_logf,ctrl_logl)
ctrl_cmdl = '{0}␣{1}␣{2}␣{3}'.format(ctrl_exec,ctrl_args,ctrl_logp,ctrl_addr)
ctrl_out = open('output/of-out-{0}'.format(start_time),'w')
ctrl_err = open('output/of-err-{0}'.format(start_time),'w')
ctrl_popn = c0.popen(ctrl_cmdl,stdout=ctrl_out,stderr=ctrl_err,shell=True)

# Set performance specifications for controller links
i1_0.config(**link_params_control)
i2_0.config(**link_params_control)

# Print out network metadata
print('Created␣network:')
print('Hosts...........{0}'.format(join_elements(net.hosts,repr)))
print('Switches........{0}'.format(join_elements(net.switches,repr)))
print('Controllers.....{0}'.format(join_elements(net.controllers,repr)))
print('Switch␣1␣ports...{0}'.format(join_elements(get_links(s1),str)))
print('Switch␣2␣ports...{0}'.format(join_elements(get_links(s2),str)))

# Execute tcpdump client on switch 1 to capture packets
interface_name      = str(i1_0)
dump_name           = 'output/of-packetdump-s1-{0}'.format(start_time)
pcap_filter         = 'tcp␣and␣port␣6633'
s1.cmd('tcpdump␣-i␣{0}␣-w␣{1}␣-U␣{2}␣&'.format(interface_name,dump_name,
    pcap_filter))

# Execute tcpdump client on switch 2 to capture packets
interface_name      = str(i2_0)
dump_name           = 'output/of-packetdump-s2-{0}'.format(start_time)
s2.cmd('tcpdump␣-i␣{0}␣-w␣{1}␣-U␣{2}␣&'.format(interface_name,dump_name,
    pcap_filter))

# Start "server" on host 2-1, simulating a webserver (without serving any
    content).
target_address      = i2_1.IP()
h2_1.cmd('netcat␣-k␣-l␣{0}␣80␣&'.format(target_address))

```

```
# This allows the controller to learn about the path to the target system.
h1_2.cmd(’netcat␣{0}␣80␣-w␣1’.format(target_address))

# This first wait allows the flow rule to expire, but the controller still
# "knows" where to find the client system.
print(’Waiting␣for␣flow␣rule␣expiry␣(1)␣-␣Duration:␣[{0}]’.format(id_wait))
time.sleep(id_wait)

# Perform two measurements. The first measurement should be slower than the
# second, as it traverses the slow path, while the second traverses the fast
# path.
res1 = probe_time(h1_2, i1_2, i2_1)
res2 = probe_time(h1_2, i1_2, i2_1)
print(’Initial␣measurements␣-␣First:␣[{0}]␣-␣Second:␣[{1}]’.format(res1,res2))

# Wait for flow rule to time out. This ensures that flow rule aggregation is
# occuring, rather than merely establishing the existence of a flow rule.
print(’Waiting␣for␣flow␣rule␣expiry␣(2)␣-␣Duration:␣[{0}]’.format(id_wait))
time.sleep(id_wait)

# Client 1-1 and 1-3 both connect to "server" 2-1. This should result in an
# aggregated flow rule being installed.
h1_1.cmd(’netcat␣{0}␣80␣&’.format(target_address))
h1_3.cmd(’netcat␣{0}␣80␣&’.format(target_address))

# Client 1-2 does a second probe, returning new times. Now, both measurements
# should be approximately equal, allowing us to conclude that an existing
# flow rule - created for other users - was installed prior to the measurement.
res1 = probe_time(h1_2, i1_2, i2_1)
res2 = probe_time(h1_2, i1_2, i2_1)
print(’Second␣measurements␣-␣First:␣[{0}]␣-␣Second:␣[{1}]’.format(res1,res2))

# Shut down "server" on host 2-1, as well as the client connections.
h1_3.cmd(’killall␣netcat’)
h1_1.cmd(’killall␣netcat’)
h2_1.cmd(’killall␣netcat’)

# Send interrupt to stop tcpdump
time.sleep(1)
s1.cmd(’yes␣|␣killall␣-int␣tcpdump’)
s2.cmd(’yes␣|␣killall␣-int␣tcpdump’)
h2_2.sendInt()
time.sleep(1)

# Terminate controller. This must be done manually, as we are using POX.
ctrl_popn.terminate()
ctrl_out.close()
ctrl_err.close()
c0.cmd(’ps␣aux␣|␣awk␣\’/pox/␣{print␣$2}\’␣|␣sort␣|␣uniq␣|␣xargs␣kill’)

# Shut network down
net.stop()
```

Listing D.8: attack-demo-id.py

D.8 tcp-time

This program is intended to be used to time the response of a target system to a TCP SYN packet, essentially a TCP version of the *ping* tool. It is used by the program in Section D.7 to perform measurements, also it can also be used standalone. It takes a measurement of the current time, sends a TCP packet with the SYN flag set using the *srp1*[7] function of Scapy, then measures the time of the response. The result is returned in milliseconds.

```python
#!/usr/bin/env python

import sys
import time
from os import popen
from scapy.all import srp1, IP, TCP, Ether, Packet

if len(sys.argv)   != 7:
    required_args  = ’<dst_ip>␣<src_ip>␣<dst_mac>␣<src_mac>␣<interface>␣<port>’
    print(’Invalid␣arguments:␣{0}␣{1}’.format(sys.argv[0],required_args))
    sys.exit(1)
else:
    dst_ip          = sys.argv[1]
    src_ip          = sys.argv[2]
    dst_mac         = sys.argv[3]
    src_mac         = sys.argv[4]
    interface       = sys.argv[5]
    dst_port        = int(sys.argv[6])

# Construct TCP SYN packet.
packet              = Ether(dst=dst_mac,src=src_mac)/IP(dst=dst_ip,src=src_ip)/TCP(dport=dst_port,flags=’S’)

# Packet transmission time.
transmission_time   = time.time()

# Perform transmission and reception
received_packet     = srp1(packet,iface=interface,timeout=5)

# Fetch reception time, outputting 0 if nothing was received.
reception_time      = received_packet.time if isinstance(received_packet, Packet) else time.time()

# Total transit time (in ms).
time                = 1000 * (reception_time - transmission_time)

# This is the result which will be parsed by the calling process
print(’Time:␣[{0}]’.format(time))
```

Listing D.9: tcp-time.py

[7]Send a packet at layer 2 and wait for a single reply

D.9 id-probe

This shell script executes attack-demo-id several times. This is to attempt to find the statistical variance of the measured times. The results are stored in a CSV file, specified in the source code. The argument is *number of runs.*

```
#!/usr/bin/env bash

# This script is intended to probe the results of an information
# disclosure (through timing analysis) attack. The script performs the
# attack COUNT times, storing the results in a CSV file.
#
# Usage id-probe <Number of runs>
COUNT=$1

# Command to execute
CMD="python␣attack-demo-id.py␣params2.json"

# Temporary file
TEMP_FILE="output/id.tmp"

# File to store results to
OUTPUT_FILE="output/id-stats.csv"

# Initialise file
echo "\"Iteration\",\"Slow path measurement with no aggregated flow\",\"Fast
    path measurement with no aggregated flow\",\"Slow path measurement with
    aggregated flow\",\"Fast path measurement with aggregated flow\"" >
    $OUTPUT_FILE

# Function to execute attack
execute_attack() {
    sudo SOFT_TIMEOUT=10 $CMD | tee $TEMP_FILE
    sleep 5
}

# Function to get latest timestamp
get_timestamp() {
    ls -l output/of-packetdump-s1-* | awk '{print $9}' | sort | tail -n 1 |
        cut -d- -f4
}

# Clear up junk
cleanup_files() {
    TIMESTAMP=$1

    # These files are generated by attack-demo-id.py
    yes | sudo rm output/of-protostat-$TIMESTAMP
    yes | sudo rm output/of-out-$TIMESTAMP
    yes | sudo rm output/of-err-$TIMESTAMP
    yes | sudo rm output/of-packetdump-s1-$TIMESTAMP
    yes | sudo rm output/of-packetdump-s2-$TIMESTAMP
    yes | sudo rm output/c0.log
    yes | sudo rm $TEMP_FILE

    # Ensure that no processes are left over
    ps aux | awk '/ofprotocol|controller|ofdatapath|ovs-controller|ovsdb-
        server|ovs-vswitchd|python|tcpdump|netcat/ {print $2}' | sort | uniq
        | sudo xargs kill

    # Remove sockets created by OpenvSwitch, if necessary
    sudo rm -rf /tmp/vconn-unix.* > /dev/null
}

```

```
# Function to compile results
process_results() {
    VALUE=$1
    FIRST=` grep ’Initial’  $TEMP_FILE | cut  -d[ -f2 | cut -d] -f1`
    SECOND=`grep ’Initial’  $TEMP_FILE | cut  -d[ -f3 | cut -d] -f1`
    THIRD=` grep ’Second m’ $TEMP_FILE | cut  -d[ -f2 | cut -d] -f1`
    FORTH=` grep ’Second m’ $TEMP_FILE | cut  -d[ -f3 | cut -d] -f1`
    MEASUREMENTS="$FIRST,$SECOND,$THIRD,$FORTH"
    echo "Measurement:␣[$VALUE]␣-␣Values:␣[$MEASUREMENTS]"
    TOTAL=`echo "$FIRST␣+␣$SECOND␣+␣$THIRD␣+␣$FORTH" | bc | cut -d. -f1`
    if [[ $TOTAL -gt 5000 ]]; then
        echo "Excluding␣measurement..."
    else
        echo "Including␣measurement..."
        echo $VALUE,$MEASUREMENTS >> $OUTPUT_FILE
    fi
    TIMESTAMP=`get_timestamp`
    cleanup_files $TIMESTAMP
}

# Initialise counter
VALUE=1

# Main loop
while [[ $VALUE -le $COUNT ]]; do
    echo "Attempting␣to␣perform␣attack␣-␣Iteration:␣[$VALUE]"
    execute_attack
    process_results $VALUE
    let "VALUE␣=␣VALUE␣+␣1"
done
```

Listing D.10: id-probe.sh

Appendix E

Timetable

Date	Description	Overview
15.10.2012	Initial meeting	Define methodology
19.10.2012	First meeting; Attack model, discussion of virtualisation	
26.10.2012	Discuss methodology; discuss timetable	
02.11.2012	Finalise methodology?	
09.11.2012		Create DFD
16.11.2012		
23.11.2012		Create attack tree(s)
30.11.2012	First draft	
07.12.2012		
14.12.2012		Prepare presentation and report
20.12.2012	*Intermediate Presentation 1 at 14:00*	
28.12.2012		*Holidays*
04.01.2013		
11.01.2013		Experimental setup Prevention and mitigation Overview of newer specifications
18.01.2013		
25.01.2013		
01.02.2013		
08.02.2013		
15.02.2013		
22.02.2013		
28.02.2012	*Intermediate Presentation 2 at 14:00*	
01.03.2013		
08.03.2013		
15.03.2013		FlowVisor
22.03.2013		
29.03.2013		Experimental evaluation
05.04.2013		
12.04.2013		Revision
14.04.2013	*Submit report*	Presentation

Date	Description	Overview
19.04.2013	(No meeting)	
25.04.2013	*Final presentation at 15:15 (CSG meeting)*	