VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Detecting and mitigating DNS amplification attacks using SDN

Ben Vidulich

Supervisors: Qiang Fu, Andy Linton, Dean Pemberton, Nemir Shaker, Ewen McNeill

Submitted in partial fulfilment of the requirements for Bachelor of Engineering.

Abstract

DNS amplification attacks have become a prevalent means of taking down websites or servers on the internet. Although there are many possible solutions to mitigate the attacks the solutions are not widely implemented and few solutions have the capability of operating on internet-exchange (100GBps+) levels of traffic. This report presents a new approach to mitigating the attacks that aims to succesfully operate on high-traffic networks and effectively reduce the number of attacks without the need for widespread adoption.

Acknowledgments

I would like to my supervisors at VUW — Qiang Fu, Andy Linton, and Dean Pemberton — for their advice and support during the project. Dean also contributed the idea that became the design that was used during the project.

I would also like to thank Nemir Shaker at CityLink for his support as my industry supervisor, and Ewen McNeill and Nic Cave-Lynch for their assistance during the industry project.

The project was inspired by the blog posts of Matthew Prince at CloudFlare.

Contents

Figures

Chapter 1

Introduction

The internet, a tool originally created to make it easier for researchers to collaborate on their work, has become a prevalent part of society that has extended far beyond its original intentions. E-commerce, social networking, gaming, blogging, and telecommunications are a small number of common uses of the internet today. The original designers of the internet made the assumption that everyone connected to the internet could be trusted, however that is no longer the case. Attackers are taking advantage of this poor assumption to cause chaos for innocent (and other malicious) participants of the internet.

An especially common goal of internet attackers is to force websites offline for political or financial reasons. Attackers use Distributed Denial of Service (DDoS) attacks to achieve their goals. A DDoS attack can force a website offline by consuming all of the resources of the computers that host the website. While a victim web server is trying to process and respond to the bulk requests of the attacker it cannot do the same for the requests of legitimate users. Figure 1.1 shows how DDoS attacks are widespread, varying largely in origin, destination, bandwidth, and attack type.

1.1 Problem statement

DDoS attacks are largely possible due to poor router hygiene and server misconfiguration. Many routers do not check the source addresses of the packets they forward, allowing attackers to fake the origin of malicious content they send. Responses to these *spoofed* requests are then delivered to an unsuspecting recipient. Servers are misconfigured to respond to requests from anyone on the internet when it is likely that they were intended to be used by a smaller number of users. It is suggested that administrators of these servers are unaware of the risks of configuring their servers to allow open access, or they do not know how to lock down the configuration on their servers. Another potential problem is with servers that run out-of-date versions of software: vulnerabilities in the servers are actively exploited by attackers and this can lead to more potential unwilling participants of DDoS attacks.

A solution is required that will dampen the effect of DDoS attacks while requiring minimal change to existing internet infrastructure. It is to be assumed that server administrators will not fix their misconfigured servers or routers, or keep server software up-to-date. Therefore, the solution should be placed under the control of proactive administrators in key locations across the internet. These key locations will have large amounts of regular traffic flowing through them and the volume is likely to increase as additional uses for the internet become available, so the solution should be highly scalable.

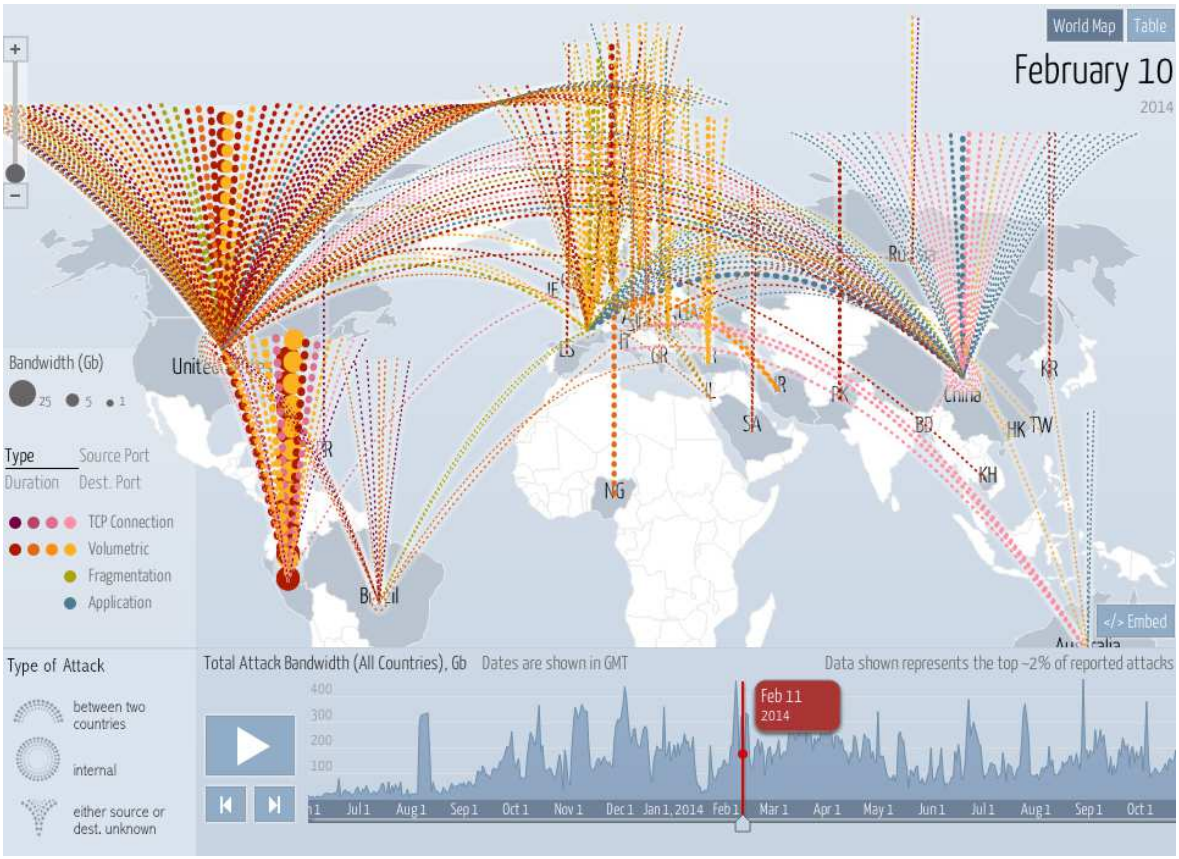

Figure 1.1: Visual representation of DDoS attacks recorded by Arbor Networks on February 10 2014 [19]

1.2 Objective

The objective of this project to provide a viable mechanism to mitigate DDoS attacks.

- The solution should be capable of handling an internet exchange-level volume of traffic.
- The solution should run on a variety of hardware or software.
- The solution should be extensible in case other attacks need detecting.
- The solution should be able to mitigate attacks with minimal impact to legitimate traffic.
- The solution should be effective without requiring modifications to every router on the internet.

1.3 Contributions

The project provides contributions in the following areas:

- An analysis of possible solutions that could support a large volume of regular traffic and attack traffic.

- The design and implementation of a mechanism for detecting amplification attacks.
- The design and implementation of a mechanism for mitigating amplification attacks.
- The development of a mechanism to test the effectiveness of an amplification attack detection and mitigation system.

1.4 Note

There was a risk that the original industry-based project (titled *Bringing the world's first SDN controlled Internet Exchange into production*) would not be completed in time due to legal and organisational issues. To mitigate the risk a plan was established for an alternative project that would continue to use SDN but to solve a different problem. The risk occurred and what follows in this report is the outcome of the alternative project.

1.5 Organisation of the report

The rest of the report is organised as follows:

Chapter 2 describes in detail the components of a DNS amplification attack, potential approaches to mitigating DNS amplification attacks, and introduces Software-Defined Networking (SDN). Then it explores real-world examples of amplification attacks.

Chapter 3 compares potential approaches to solving the problem and describes in depth the approach that is taken to solve the problem in this project.

Chapter 4 describes how the solution has been implemented.

Chapter 5 evaluates the effectiveness of a solution in terms of the project's requirements.

Chapter 6 concludes the project and discusses future work that could be performed on the project.

Chapter 2

Background and literature review

This chapter seeks to dissect and explain the components of a DNS amplification attack and investigate methods from industry and academia to detect and mitigate these attacks. Also included in this chapter are case studies of DNS amplification attacks occurring in the real world.

2.1 Components of a DNS amplification attack

An attack is usually constructed of multiple fundamental components that exploit weaknesses in a system. This section will describe the components that form a DNS amplification attack.

2.1.1 Distributed denial of service attack

A Distributed Denial of Service (DDoS) attack can be broken into two parts: *distributed* and *denial of service*. Fundamentally, a denial of service attack is where legitimate users' access to a service is impeded by the attempts of an attacker [22]. If the attacker is successful then no legitimate access to the service is possible. A common method used by attackers to deny service to legitimate users is to consume the resources of the service so that the resources cannot be used to serve legitimate users.

The attacker can consume a greater amount of the service's resources by attacking with more resources — thus denying the service to more users. Increasing the resources in an attack is achieved by having several computers distributed across the internet [35]. A non-distributed attack can generally be mitigated by blocking the IP address of the attacker. However, this becomes difficult with a DDoS attack because there is an overwhelming number of attack sources — blocking individual IP addresses of attackers becomes time-consuming, new attack sources may appear, and it may be difficult to distinguish between attackers and legitimate users. A common technique to overcome DDoS attacks is to *black-hole* the IP address(es) being targeted. Blackholing will be discussed further in section 2.3.1.

An attacker distributes a DoS attack by taking control of unsuspecting users' computers and turning them into zombies that can be controlled remotely [35]. A network of remote-controlled zombie computers is known as a *botnet* (robot network) [15]. Each computer in a botnet is usually infected with some sort of *malware* and is then known as a zombie. A zombie can be instructed to perform a number of tasks including but not limited to sending spam, clicking advertising, and contributing to DDoS attacks.

2.1.2 Reflection attack

A reflection attack is a sub-component of an amplification attack (see section 2.1.3). Suppose an identification protocol uses a challenge-response system where one principal sends a challenge and the other principal proves their identity by sending the correct response. Now suppose that an attacker wants to break the protocol by posing as a legitimate principal. When the challenging principal sends a challenge to the attacker (equation 2.1) the attacker reflects the challenge back at the challenging principal (equation 2.2). Assuming the protocol is symmetrical and the challenging principal is instructed to respond to any challenges it receives then it will reply to the reflected challenge with a valid response (equation 2.3). The attacker then reflects back to the challenging principal (equation 2.4) and the challenging principal treats the attacker's identity as a valid principal. This is known as a *reflection attack* [1].

$$A \rightarrow B : N \tag{2.1}$$

$$B \rightarrow A : N \tag{2.2}$$

$$A \rightarrow B : \{N\}_K \tag{2.3}$$

$$B \rightarrow A : \{N\}_K \tag{2.4}$$

The weakness in the aforementioned protocol is the symmetric behaviour of the challenges and responses. A simple way to mitigate this attack is to make the protocol asymmetric: for example, varying the challenges and/or responses with each run of the protocol. Furthermore, each response could include the identity of the principal sending the message (equation 2.6) [37]. Therefore, a reflection attack can be detected if a principal receives a challenge that appears to come from itself.

$$A \rightarrow B : N \tag{2.5}$$

$$B \rightarrow A : \{B, N\}_K \tag{2.6}$$

2.1.3 Amplification attack

The aforementioned reflection attack pertains mostly to authentication systems, however the same concepts can be applied to form the basis of an amplification attack. The goal of an amplification attack is to attack the target with more resources than are directly available to the attacker. The goal relates to a reflection attack because an attacker can exploit a protocol that does not verify the identity of a request before sending a response. Both ICMP and UDP are stateless by design and therefore these protocols can be used to reflect a response to a destination host that is not the same as the request packet's origin host. This can be useful for an attacker to protect their identity when performing an attack, however the attack becomes more valuable to the attacker when the reflected response is larger than the request made by the attacker. For example, the Smurf attack in section 2.1.3 responds with more packets that were originally sent and the attacks in section 2.1.3 response packets that are larger than their request packet counterparts.

Smurf attack

The first known instance of an amplification attack is called a Smurf attack. `smurf.c` was created in 1997 by a user with the handle *Tfreak* [28]. The attack works by sending an ICMP request packet to the broadcast address of an unsuspecting network that has a gateway router that is configured to relay ICMP requests to devices sitting behind the router [31].

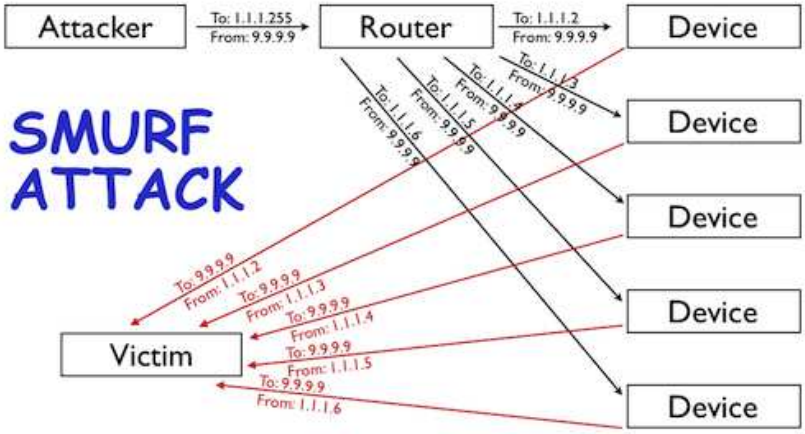

Figure 2.1: Example of a Smurf attack [31]

The source address of the ICMP request packets is spoofed to match the IP address of the target host. When the unsuspecting hosts send their ICMP replies they are directed at the target host. The amplification factor of a Smurf attack depends on the number of hosts that receive the ICMP request sent to the broadcast address. For example, in figure 2.1 there are five hosts behind the poorly-configured router which means the amplification factor is 5 times.

Smurf attacks are easily mitigated by configuring routers not to forward ICMP requests to the broadcast address of a network [31]. Alternatively, individual hosts could be configured to ignore ICMP requests but this may prevent purposeful network diagnostics.

UDP-based amplification attacks

Tfreak is also known to have created the first UDP-based amplification attack, called Fraggle after its file name `fraggle.c` [28]. Like ICMP, UDP has no handshaking mechanism and no other way to verify the source IP address of a packet. Therefore, weaknesses in publicly-accessible servers that use the UDP protocol can be used to launch amplification attacks [24]. Instead of relying on the number of response packets to generate an amplification factor, an attacker can send a small UDP request packet that generates a much larger UDP response packet. Figure 2.1 shows several common services that operate over UDP and their associated Bandwidth Amplification Factor (BAF). BAF, as shown in equation 2.7, measures the size of the response payload compared to the request payload. Figure 2.1 lists CharGEN as the protocol with the highest bandwidth amplification factor. CharGEN is a mostly unused [2] protocol that generates a stream of assorted characters over a network socket. Over UDP the response payload size can be up to 512 bytes depending on the size specified in the request.

$$\text{B.A.F.} = \frac{\text{payload size}(response)}{\text{payload size}(request)} \tag{2.7}$$

2.2 DNS amplification attack

DNS amplification attacks are a prime attack vector for attackers: they support queries over UDP - allowing the source address to be spoofed - and they have a high bandwidth ampli-

Protocol	Bandwidth amplification factor
DNS	28 to 54
NTP	556.9
SNMPv2	6.3
NetBIOS	3.8
SSDP	30.8
CharGEN	358.8
QOTD	140.3
BitTorrent	3.8
Kad	16.3
Quake Network Protocol	63.9
Steam Protocol	5.5

Table 2.1: Bandwidth amplification factor (BAF) of common UDP-based services [24]

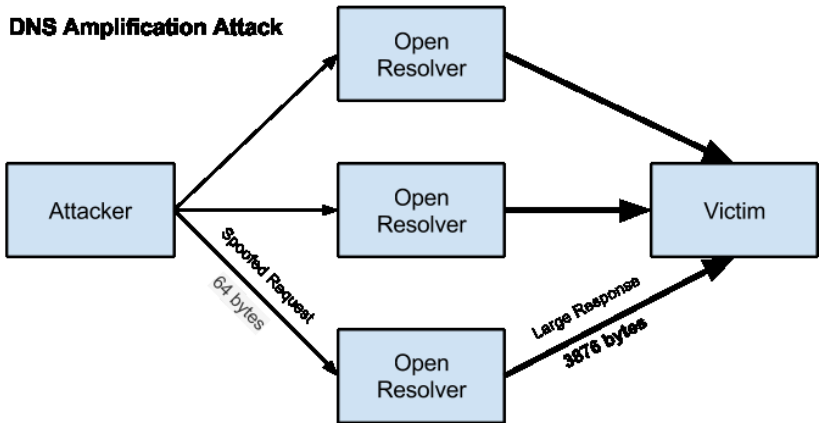

Figure 2.2: Anatomy of a DNS amplification attack [5]

fication factor (see figure 2.1) [31].

The attack takes advantage of DNS resolvers that accept and respond to queries from anyone on the internet. These resolvers are known as *open resolvers*. The open resolvers respond to the spoofed queries with responses that are significantly larger than the attacker's spoofed requests (see figure 2.2). Each open resolver typically has a large available bandwidth which allows the attacker to continue to use the resolver without the resolver denying itself service. The combined available bandwidth of the open resolvers is much larger than the attacker could provide with just their own internet connection. Section 2.4 will discuss real-world examples of massive-scale DNS amplification attacks and the impact of each attack on victim networks.

Matteo Cantoni presents the results of his DNS honeypot on his website [4], which show statistics about target IPs, target countries, and the Fully Qualified Domain Names (FQDNs) used in the attacks. FQDNs listed on Cantoni's website have response payloads ranging from 35 bytes to 37776 bytes. Assuming each query has a size of 64 bytes, the amplification factor ranges from 0.5 to 590. An example of a large DNS response is shown in appendix A. The Linux utility *dig* is used to request `doc.gov`, which returns an 8230 byte response.

2.3 Mitigating DNS amplification attacks

There have been various attempts to mitigate DNS amplification attacks - some attempts are temporary measures to withstand specific attacks, while other attempts try to permanently mitigate attacks. This section will cover some of the common techniques for mitigating these attacks.

2.3.1 Blackholing

The act of *blackholing* is to instruct a router to drop incoming traffic before it reaches the destination network [18] [34]. This approach allows temporary relief from an incoming DDoS attack by preventing attack traffic from reaching critical components on a network. A more specific definition of a blackhole is a configuration where incoming IP packets destined for a specific host or network are forwarded to a null queue (thus dropping the packets). The disadvantage of this technique is that all incoming traffic is dropped, including traffic from legitimate users. If the attacker's target has multiple redundant networks then the target could null route (blackhole) incoming attack traffic on one network and direct legitimate traffic to a different site or network.

2.3.2 Source Address Verification

Since all amplification attacks (and many DDoS attacks in general) depend on the ability to spoof the source IP address of a packet, there have been attempts to prevent attackers from using this ability such as BCP 38 [9]. BCP 38 filters incoming packets by verifying that it is possible to reach the network of the packet's source IP address through the interface on which the packet was received [23]. For example, a router on the edge of an ISP's network checks the source IP address of any incoming packets. If the source IP address of the packet matches a network prefix that is known and known to originate from that interface then the packets are allowed to enter the ISP's network. Figure 2.3 demonstrates the actions of BCP 38 during a DDoS attack: forged packets from the attacker's network are dropped while packets legitimate packets from the attacker's network are allowed into the ISP's network.

A criticism of Source Address Validation (SAV) is that network operators must all implement SAV for the technique to be effective [43]. Enabling SAV on all networks would be an expensive exercise and given the number of network operators and devices connected to the internet it would be difficult to ensure every device supports and enables SAV. Mandating SAV would be difficult, if not impossible, because there is no way to determine from the outside whether SAV is active and there is no business case for a network operator to have an audit performed internally.

Furthermore, Vixie writes that SAV would not directly solve the problem that causes DDoS attacks. The SAV technique assumes that forged packets have a source IP address that is invalid in context (for example, a RFC 1918 address) or the router receiving the packet would normally use a different interface to sent packets to that source IP address. However, the target IP address of a DDoS attack might appear to be valid if the unsuspecting router uses that same interface or routing path to send packets to the target IP address. For example, an attack being bounced through New Zealand to and from other countries may be undetected by SAV due to the limited number of paths a packet can take to enter or leave New Zealand.

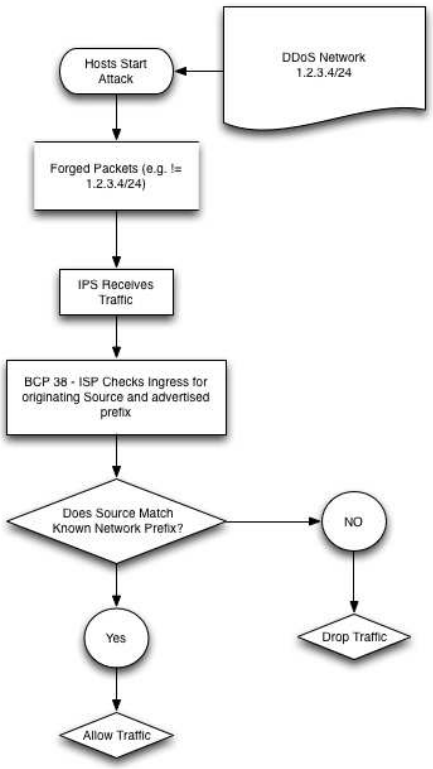

Figure 2.3: Simple workflow of BCP 38 during a DDoS attack [30]

Ingress filtering has been suggested by authors of many academic papers [6] [13] and has become a best current practice (BCP 38 [9]) for the internet community. As mentioned in section 2.3.2, routers compare the source addresses of incoming packets against their forwarding tables to ensure the packets come from known prefixes and are allowed to be received on that interface. For example, a router may be configured to only allow packets from x.x.y.y/zz on its ingress link. The router would check the source address of incoming packets and if the source address does not match the prefix x.x.y.y/zz then the source address is deemed invalid and the packet is dropped. Kong et al. [13] argue that a router must know which prefixes are allowed to send traffic to the router's incoming interface. This may be the case when the router is used to peer with another network, thus access to the network is provided to a limited number of hosts, but if the router is used for upstream transit then incoming packets could effectively come from any other host on the internet. Therefore, this technique is more effective if all routers on the internet implement ingress filtering since the filtering is performing at the source — before the potentially spoofed packets reach the rest of the internet.

2.3.3 Disabling recursion on authoritative name servers

Most authoritative name servers are intentionally made internet-accessible so that the domain names for which these name server have authority can be queried by public internet users. Since most DNS server applications can perform both authoritative functions and recursive functions it is possible that an authoritative name server unintentionally provides recursive functionality to anyone who queries the DNS server, including public internet users. This kind of DNS server is called an open resolver and is described in more depth in section 2.2.

The recommended action to prevent an authoritative name server from being used as an open resolver is to disable the recursive functionality on the name server [23] [41]. The most common use of an authoritative name server is in a public internet environment where there is no need for recursion to be enabled. On the other hand, authoritative name servers in a private network environment may need to use both functions at once. For example, in a Split DNS configuration [7] the internal name server might be configured to resolve domain names to RFC 1918 [36] addresses and provide recursive functionality to hosts in the network, while the external name server might be configured to resolve domain names to public IP addresses. In this example it would be acceptable to allow recursion on the internal authoritative name server but the external authoritative name server should have recursion disabled to prevent the name server from being used in a DDoS attack.

2.3.4 Limiting recursion to authorised clients

Instead of completely disabling recursion (as in section 2.3.3), DNS servers can be configured to allow queries from certain blocks of IP addresses [23] [5]. For example, an ISP may wish to only recursively resolve queries from their own customers. However, even a DNS resolver that is limited customers can be susceptible to DDoS attacks as explained in section 2.4.3.

2.3.5 Response Rate Limiting

A relatively new approach to mitigating DDoS attacks is a methodology called DNS Response Rate Limiting (DNS RRL) [42]. The methodology is based on the idea that a caching resolver should not need to request a resource record (RR) more often than the time-to-live (TTL) of the RR. In other words, the resolver counts number of times queries are received from a given source address for a specific RR within a given time frame. If a counter reaches a certain threshold within the specified time frame then subsequent requests from that source address for that RR are not replied to. Spoofed packets will all have the same source address and will most likely be requesting the same RR too. Replies to these requests will quickly become limited. It is highly unlikely that a legitimate user would request the same RR within the given time frame, due to caching, and therefore it is unlikely that a legitimate user would be affected by the rate limiting. However, if a legitimate user's queries are dropped due to rate limiting then the user can try again over TCP, since TCP requests are harder to spoof, or send their request to a different resolver.

Figure 2.4 shows the incoming and outgoing traffic to a DNS root zone server before and after DNS RRL was implemented on the server. Prior to the implementation of DNS RRL the amplification effect is clearly demonstrated. After the attacker gives up, it becomes clear that the attacker's traffic composed most of the input traffic to the server but did not compose a large portion of the output traffic. This suggests the attack became highly ineffective after DNS RRL was implemented.

Vixie provides an economic viewpoint in [43] when he describes a DDoS attack: an attacker seeks to minimise their cost while maximising their utility. This means that an attacker will try to reduce the amount of traffic they need to send themselves to cause the greatest impact to their target. By design, DNS RRL limits the number of responses it sends in a given time frame. This means that an attacker that is reflecting traffic through a server that has implemented DNS RRL will send less traffic at the target than the attacker would by directly sending traffic at the target. This attenuating effect in theory should discourse the attacker from using the hardened resolver in their attack. If enough resolvers implement

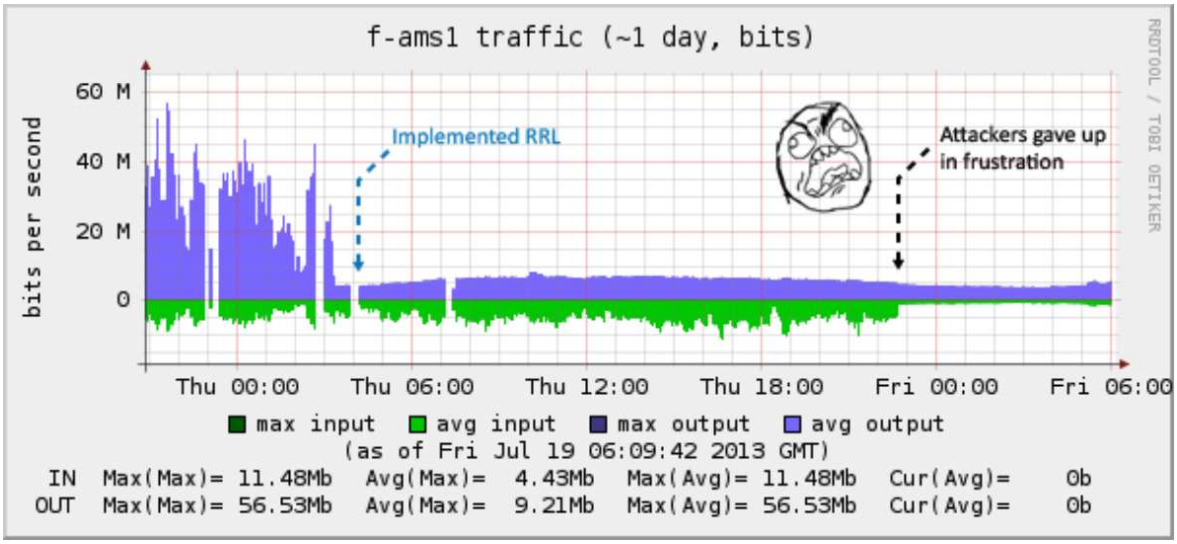

Figure 2.4: Graph displaying traffic before and after deploying DNS RRL to a root DNS server [44]

DNS RRL then the attacker's efforts are completely thwarted leaving the attacker to locate weaker resolvers, to try a different kind of attack, or to give up.

DNS RRL is intended to be implemented in DNS server software or in an intrusion detection system which can be released as a simple software update. However, DNS RRL will be most effective if all DNS server deployments implement and enable DNS RRL. The network operators that unknowingly run open resolvers on their networks are also unlikely to know about the DNS server software updates, how to configure DNS RRL, or why it is important. Therefore, while the diligent network operators will be less susceptible to being part of DDoS attacks, the incompetent or ignorant network operators will continue to allow DDoS attacks to flow through their networks.

2.3.6 Reactive DDoS defence

Kong et al. [13] propose adding tags to IP packets involved in DDoS attacks so that the packets can be traced back to their origins. Each router adds a tag to each packets as a part of the regular forwarding process. Once a packet reaches its destination the exact path taken by the packet can be traced back to the packet's first hop. Packet marking defeats UDP and ICMP packet source address spoofing because marks are added to the packets after they have left the origin. On the other hand, this technique assumes that routers can be trusted to append the correct tag and that routers have implemented the packet-marking code. An attacker could spoof the packet markings which would increase the complexity of a traceback. Furthermore, the authors' suggestion that spoofed packets be allowed to consume router resources prior to reactive control could effectively deny service to legitimate traffic passing through the router when there is a large volume of spoofed packets passing through the router. Source trackback would be most effective if all internet routers implemented packet marking, however this would be economically and technically infeasible in achieve. Regardless, if routers close to the origin implemented the protocol — or routers transferring the request packet of an amplification attack — then that would be enough to estimate the origin of the attack. Appending a mark to a packet would extra computations in a router, which would slow the packet processing rate of the router. Also, the more hops a marked packet passes through the longer the packet size which would increase the trans-

mission delay between hops and require packet segmentation if the mark field dynamically expanded.

Attack traffic could be throttled instead of filtered, according to Mahajan et al. [14]. When a router detects a spike in traffic it can ask its upstream router to throttle the traffic being sent to it. The spike in traffic is detected by measuring the loss rate of a link and when a significant loss is sustained for a certain length of time then a spike in traffic is said to have occurred. Mahajan et al. suggest historical loss rate values as a baseline to distinguish between regular and irregular amounts of congestion. Kong et al. [13] criticise this approach, suggesting that this mechanism may be ineffective against an aggressive attack with multiple attack flows. This would make the mechanism ineffective to operate on a router that is situated near to the target of a distributed attack. The pushback mechanism is described to ask upstream neighbours that are sending presumed attack traffic to throttle the amount of traffic being sent to this router. This approach would be effective assuming acknowledgements do not get lost on the congested link and the upstream router also supports the pushback protocol. Mahajan et al. describe the pushback mechanism as optional, however pushback would be extremely useful for routers near an attack target. There is also a traffic throttling element for the router under attack. This element functions very similarly to common Quality of Service (QoS) implementations but with a dynamic approach: attack traffic traffic is directed onto a virtual queue that, like a normal queue, drops packets when the queue is full. The advantage of this approach is that the virtual queue fills before the normal queues and therefore only attack traffic is dropped, however the implementation of the virtual queue will require extra memory and more computations along the forwarding path.

Sun et al. [39] focus on an efficient method to distinguish between and filter legitimate DNS traffic and attack DNS traffic. Their detection scheme focuses on the traffic ingress and egress to an ISP: any incoming or outgoing DNS request should have a corresponding response in the other direction. If there is a significant imbalance of incoming and outgoing DNS packets then an attack is most likely occurring. To distinguish between DNS requests and responses Sun et al. focus on UDP traffic with either a source or destination port of 53. A request packet will have a destination port of 53 and a response packet will have a source port of 53. The DNS response flag is also set to *1* for a response packet. The number of requests and responses passing through the router is compared and the value is passed through a low-pass filter for smoothing. If the output of the filter is greater than zero then an attack is present. Once the presence of a DNS amplification attack has been detected Sun et al. pass DNS response traffic through their two-bloom filter. The filter scheme works by storing a *4-tuple* (source IP, destination IP, source port, DNS transaction ID) in one of the two bloom filters. All DNS requests collected in the first time interval are stored in the first bloom filter and all DNS requests collected in the second time interval are stored in the second bloom filter. The bloom filters are flushed in an alternating pattern at the end of a time interval so that DNS requests for at least the last time interval and at most the last two time intervals are stored. When an incoming DNS response is received its *4-tuple* is looked up in both of the bloom filters. If a matching *4-tuple* was found in a bloom filter then there is a high probability that the response is legitimate. Bloom filters have the possibility of returning false positives and therefore there is a small chance that attack traffic will be treated as legitimate traffic, however a majority of the attack traffic will be filtered. Sun et al. demonstrate the effectiveness of their solution on a link with a rate of 39.8 GBps, which would allow the solution to implemented by medium-sized transit providers with little demand for memory and little CPU overhead. This solution is DNS specific making it difficult to expand to other UDP-based attacks, however it could be a highly preventative mechanism without needing every router on the internet to implement the solution.

2.4 Case studies

This section describes examples of real-world attacks with large attack vectors or with large implications.

2.4.1 DNS amplification attack on Spamhaus

In the first quarter of 2013 not-for-profit anti-spam organisation Spamhaus was knocked offline by a DNS amplification attack that peaked at over 100 GBps of amplification traffic. Mimososo writes in Threatpost [16] that users of the Dutch webhost Cyberbunker were unhappy to be flagged as spam and they retaliated with this attack, which brought down the Spamhaus website. This attack matches all the common characteristics of a DNS amplification attack: many open resolvers (over 30,000 resolvers recorded during the attack), spoofed DNS query packets (36 bytes in length), and massively amplified DNS response packets (3,008 bytes in length; 100x amplification factor) [33].

2.4.2 400 GBps NTP amplification attack on CloudFlare

CloudFlare was also subject to an attack of similar volume in February, however this attack reached nearly 400 GBps of attack traffic. This attack exploited a misconfiguration in NTP servers, however NTP amplification attacks operate in basically the same way as any other UDP-based amplification attack: source address is spoofed and response packet is larger than the request packet. Prince notes in his blog post [32] that open NTP servers are less common than open DNS resolvers, however NTP servers tend to be more powerful machines with higher available bandwidths, and NTP responses can be much larger than DNS responses which increases the amplification factor per response. Therefore, fewer NTP servers are needed to cause the equivalent amount of damage as a DNS amplification attack. The advantage for an attacker to require fewer servers to perform the attack is fewer machines are needed to send the spoofed requests. Prince suggests the attacker could have used a single server to send spoofed requests to the 4,529 NTP servers involved in the 400 GBps attack.

NTP amplification attacks are possible due to a specific command that can be issued against an NTP server called `MON_GETLIST` [20] [10]. The command is used to get a list of machines that have interacted with the NTP server. The maximum list size is 600 entries, which is transmitted as a 48 kB response. This equates to an amplification factor of 206 times. Server administrators can prevent their NTP servers from participating in NTP amplification attacks by disabling the `MON_GETLIST` command on their NTP servers or by limiting the IP addresses that may execute the command. Prince argues that the command serves little purpose and therefore there is no need for the command to be enable at all [32].

2.4.3 DNS amplification attack on Spark New Zealand

Internet service provider Spark experienced an outage to their DNS infrastructure in September 2014 when the infrastructure came under attack by their customers' devices. Spark claimed [12] the attack was targeted at servers in Europe, however the attack overwhelmed Spark's DNS servers instead. While Spark did not appear to be certain of what caused the attack their infrastructure the two most likely hypotheses were that their customers were affected by malware — based on the presence and behaviour of malware in other parts of the world at that time — and open DNS resolvers on some of their customer's modems.

Although Spark do not describe the exact cause of the attack (see appendix F), it is reasonable to assume that a significant amount of DNS traffic reached Spark's DNS servers and

overwhelmed the servers. It is possible that malware infected customers' devices and the malware either intentionally used Spark's DNS servers to amplify the attack or the malware creator forgot to specify the open resolvers to use and the devices infected with the malware sent the DNS queries to Spark's DNS servers by default. If customer modems were configured as open DNS resolvers then both scenarios are applicable to the modems too. Another possibility is that the modems were configured as DNS relays rather than recursive resolvers and therefore DNS queries received by the modems were automatically relayed to Spark's DNS servers.

2.5 What is software defined networking

The definition of software defined networking (SDN) is a widely explored area with many different answers. The two most common definitions of SDN are: the separation of the data plane from the control plane, and the centralisation of control [25]. Control of an SDN usually is usually programmatic; rather than having network functions defined in vendor-controlled firmware the control is written in software and can be updated more or less on-the-fly and deployed to multiple network devices at once. The separation allows multiple network devices to be controlled from the same controller which makes administration easier and reduces the demand for complex network devices. Since the controller is written entirely in software it can operate in a physical or virtualised environment as necessary. Since the network devices become commoditised: hardware becomes interchangeable and the network functions can even run on commodity virtualisation hardware just as the controller functions can.

SDN allows a switch, or any other network function, to be customised beyond the customisation-level provided by a typical network device vendor. Economies of scale makes it costly for a vendor to produce small numbers of highly-customised products and therefore product customisation is generally limited. The ability to customise a switch is especially important to this project because the ability to add additional logic to a switch with minimal impact to performance makes it more difficult for security features such as DDoS prevention to be circumvented and allows the additional logic to run in high traffic volume environments where a classic firewall would not be feasible.

2.6 Summary

There are several examples of approaches from industry and academia to solve the problem of DNS amplification attacks, however the attacks are still prevalent as demonstrated with the case studies. The ideal solution is to educate system administrators on how to correctly configure a DNS server so that it cannot be used as an open DNS resolver, and then to fix the configuration of all the open DNS resolvers on the internet. However, this is a unrealistic solution due to the large number of server administrators that needed educating and the large number of open DNS resolvers that need locking-down. Although the approaches to solving this problem in other ways are numerous, none of them seem to have made a significant impact on reducing the number of DNS amplification attacks. Many of these solutions are only applicable to small networks or require installation on a large number of devices across the internet. A new solution is needed that can operate on a small number of devices in large networks and have a large effect.

Chapter 3

Design

This chapter evaluates potential approaches to solving the problem and then describes the selected outcome as a high-level overview.

3.1 Possible approaches

This section describes a selection of approaches that could be used to detect and mitigate DNS amplification attacks. Parts of the approaches are discussed in more depth in section 2.3.

3.1.1 DNS Response Rate Limiting

A system that implements DNS Response Rate Rimiting (DNS RRL) keeps a record of the number of time times a given source address requests a certain resource record (RR) [42]. If the source address requests the RR too many times within a specified amount of time then the system stop responding to request from that address for that particular record. If a legitimate request for the record becomes limited then the requester can switch to using TCP instead of UDP, which is not prone to source address spoofing, or request the record from another name server. DNS RRL is discussed in more depth in section 2.3.5.

In a DNS server

DNS RRL has already been implemented in the DNS server program BIND and there is evidence that this technique is effective at mitigating DNS amplification attacks [44], so there is little possibility for additional contribution in this manner. However, this solution does not address all of the requirements of the project. For example, DNS RRL is not directly extendible to other kinds of DDoS attacks. Each other attack type, for example NTP amplification attacks, would require a separate design and implementation. Although it is possible, it is unlikely that any hardware-based DNS server implementations would get patched for DNS RRL due to number of different models and the time-consuming firmware updates that would need to take place.

With SDN

Instead of implementing DNS RRL directly on DNS servers it could be implemented in switches at key internet locations using SDN — operating in almost exactly the same way. The benefit is that more traffic passes through the switch and therefore there is more opportunity to detect attacks. Also, it is harder for an attacker to circumvent the mechanism

because the attacker most likely cannot affect the actions of a layer two device. On the other hand, the switch cannot inspect the contents of a DNS packet and therefore every DNS packet would need to be sent to the controller. The controller would have to inspect the packet and send legitimate packets back to the control plane. The link between the switch and the controller may not be able to handle the load of the DNS traffic and it could add unnecessary load the CPU of the controller. If the controller overloaded then it may not be able to respond to messages from switches, which would cause further disruption to legitimate traffic. The process of sending DNS packets to and from the controller would add extra latency to the delivery of DNS packets which would not meet the requirement of the project that the detection and mitigation mechanism must mitigate attacks with minimal impact to legitimate traffic. Similarly when applied to a DNS server, the DNS RRL implementation using SDN would scale well to other kinds of similar attacks. Although SDN would be it easier to extend the mechanism to other attack types the extension itself would add more load to the system.

3.1.2 Detect attacks using an intrusion detection system

Network Intrusion Detection Systems (NIDS) by definition are designed to detect attacks on a network. Traffic on the edge of a network could be passed through an NIDS to detect the attack and then a firewall or Intrusion Prevention System (IPS) could be used to mitigate the attack. Kambourakis et al. suggest a similar approach in their paper [11]. IDSs tend to have large extendible rule sets that can detect a large number of attacks. An IDS would struggle to perform when all traffic on a large network is passed through it. The CPU would be heavily consumed and there may not be enough available bandwidth to pass all traffic through the IDS.

3.1.3 Conclusion

DNS RRL does not seem like a worthwhile approach. There is already an adequate contribution to implementing DNS RRL on DNS servers and an implementation of DNS RRL using SDN does not seem feasible. However, the use of SDN would accommodate a level of flexibility around flow control that the other solutions do not offer. An IDS would be a suitable solution in a small network but would struggle to handle the load of a large network.

3.2 Selected approach

The best parts of the possible approaches could be combined to create a solution that is extensible, can handle a large amount of traffic, and is effective at detecting and mitigating attacks. Specifically, the SDN functionality would allow for large amounts of traffic to be handled and the IDS would provide an effective detection system. An SDN switch could be used to mirror some of the traffic to the IDS at a time and the IDS could determine whether an attack is present in the mirrored traffic. If so, the switch could drop the attack traffic and let the remaining traffic pass. Otherwise, the switch could mirror a different selection of traffic to the IDS and repeat the process.

For the purposes of this project it should be assumed that the combined throughput of the data plane ports is greater than the combined throughput of the ports used for mirroring. Therefore, the solution must selectively mirror traffic to the IDS, rather than mirroring all traffic. The simplest selection is to mirror traffic on a per-port basis. However, if multiple ports need to be mirrored then this may still exceed the link capacity of the mirror port.

A potentially effective solution would be to mirror traffic on a particular port when traffic appears to originate from specific UDP ports.

3.2.1 Detection

Each port on the switch will have a dummy OpenFlow rule: each rule will match traffic entering the switch on an ingress port and then take the normal action for each packet. A typical action for a packet in a switch is the perform layer 2 forwarding. The purpose of these dummy OpenFlow rules is to provide per-port traffic statistics since each OpenFlow rule automatically collects traffic statistics.

DNS amplification attacks will be detected as an increase in traffic to a port. This will be measured in two ways: first by measuring the number of packets per second on a port. Once this rate reaches a certain threshold of slightly higher than typical peak load this OpenFlow rule or tripwire will be said to have tripped. This method will be effective at detecting the possibly of attacks during peak load

For the purposes of this project the IDS will be considered a black box. The input is a subset of the traffic traversing the switch and the expected output is indication of whether the input traffic contains packets that constitute an attack; how the IDS determines the presence of attack traffic is not relevant to the course. Furthermore, the absence of specific requirements for the IDS allows a greater choice of specific IDS product. Different administrators of exchanges may have different preferences of IDS.

3.2.2 Mitigation

When the IDS indicates that attack traffic is present, the OpenFlow controller will insert an OpenFlow rule to drop subsequent traffic. The rule should be specific enough to prevent unnecessary disruption to legitimate traffic but general enough to catch any variations in the attack traffic. On the other hand the IDS should detect any variations in the attack traffic and the OpenFlow controller can create separate rules for the variations or group the rules as necessary.

The counters associated with this new OpenFlow rule should be monitored by the controller to determine whether the attack traffic ceases. When the attack traffic ceases the OpenFlow rule should be removed to prevent disruption to legitimate users.

Chapter 4

Implementation

The primary contribution in this project is the implementation of a potential solution to detect and mitigate DNS amplification attacks. This chapter will describe the implementation in depth.

4.1 Ryu

Ryu is an OpenFlow controller that was selected to be used in the project implementation. It is a well-tested controller written in Python that is easy to build OpenFlow applications on top of [38].

4.2 Morepork

Morepork is the name given to the application that operates on top of Ryu. The primary contribution of this project is implemented as this application. The application extends upon `simple_switch_13.py`, which is an example Ryu application that provides layer-two switch functionality and operates using the OpenFlow version 1.3 protocol.

There are four distinct components within the OpenFlow application: layer-two switch, tripwire, port mirroring, and the firewall. Each of these components operate relatively independently and have been implemented as a set of layers (shown in figure 4.1). Each layer is a Python class that extends the layer below. The layer-two switch functionality is the bottom layer and forms the base of the application. The application can operate using only this layer an would simply provide the core functions of a typical layer-two switch. On top of the layer-two switch layer is the tripwire layer. This is implemented by two classes: `SimpleMonitor`, another example script provided by Ryu, and `TripwireLayer`, which interprets the port and flow statistics that are received over a data path. The interpreted data is then consumed by `MirrorLayer`, the class in the next layer up, which builds a list of ports that need to mirror incoming traffic to the IDS. This layer also sends the instructions to the switch that enable and disable the port mirroring. The next layer up is provided by the FirewallLayer class. This layer has the most independence from the other layers because its only data input is a feed of alerts from the IDS. Each time it receives an alert from the IDS it builds an instruction to drop matching traffic and sends that instruction to the datapath. The following sections describe the layers of the application in more depth.

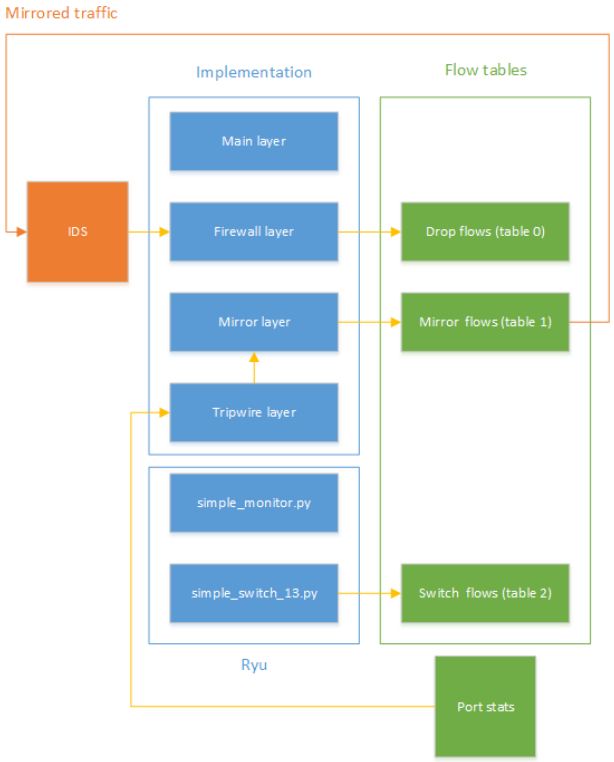

Figure 4.1: Layers of implementation and data flow to other components

4.2.1 Layer-two switch layer

The layer-two switch functionality is provided entirely by `simple_switch_13.py` and is not considered a contribution to the project.

When a packet is received at the switch the switch will look for an entry in the flow table that matches the destination Ethernet address of the packet. If a matching flow is not found then the default action is taken (defined in OpenFlow as the *table-miss* flow entry [29]). The default action defined by the `SimpleSwitch` class is to forward the packet to the controller using the OpenFlow *Packet-In* message. The controller application has a Python `Dictionary` object that stores destination Ethernet addresses as keys and switch port numbers as values. Each data path has a separate `Dictionary` object. If the destination Ethernet address of the packet in the Packet-In message is not found in the `Dictionary` object then the controller generates a *Packet-Out* message with the packet and the destination port field is set to *OFPP_FLOOD* which will cause the switch to forward the packet out all ports except the port that the packet came in on and any ports that are disabled [29]. The source Ethernet address of the packet is added to the `Dictionary`, with the port number it was received on, if it is not already there. When a host replies to the packet that was flooded the reply packet will be sent to the switch, which will record the source Ethernet address and source port, and then it will instruct the switch to forward the reply to the destination following

the aforementioned rules. If the destination Ethernet address is found in the `Dictionary` then the controller will generate a Packet-Out message instructing the switch to forward the packet on the port specified in the value of the key-value pair in the `Dictionary`. In this instance the controller will also send a *Flow-Mod* message to the switch instructing the switch to add a flow entry that matches any subsequent packets with that corresponding destination Ethernet address and to forward it to that corresponding port. Any subsequent packets destined for that Ethernet address will match the flow entry and will be forwarded on the appropriate port without flooding or sending a Packet-In message to the controller.

Common (non-OpenFlow) switches are know to perform more functions than those described above. Although it is not relevant to this project that additional features be provided, it is possible to extend the core switch features available with the project's implementation by replacing `simple_switch_13.py` with another, more advanced script. The Ethernet address flow entries are inserted into the flow table with an ID of 2. OpenFlow specifies that packets be processed in order 0, 1, ..., n. The layers of the implementation that are above this layer utilised tables with IDs 0 and 1, which means that the flow entries inserted by the highest layer will be processed first and the layer-two switch layer flow entries will be processed last.

4.2.2 Tripwire layer

This is the component of Morepork that detects anomalous spikes in traffic. As described above, this layer is implemented as two classes: the `SimpleMonitor` class and the `TripwireLayer`. The former class is a modified example script that was provided by Ryu; the class provides helper methods to request and receive per-port and per-flow statistics. Statistics are requested using a separate thread that executes an infinite loop. The loop sends `ofp_flow_stats_request` and `ofp_port_stats_request` messages along each datapath and then sleeps for ten seconds. The sleep time was selected for performance reasons that are discussed further in section 5.3 of the evaluation.

When a `ofp_port_stats_reply` is received the data is dumped into a class called `PortStatsCollector`. The class simply stores the data in a multi-dimensional `list` and provides methods for retrieving the data later. Options for data retrieval include returning the last value for a given metric name for a port on a datapath, and first and second-order derivatives based on the last two values for a given metric name for a port on a datapath. The derivative methods are used to provide rate and acceleration values, if necessary, for the threshold calculation system.

The threshold calculation system is called *Tripwire* and is located in the `Tripwire` class. This class reads a list of threshold values from a configuration file and compares them to data from an instance of the `PortStatsCollector` class. The configuration file stores data in the YAML format [8], as shown below. At the top level of the configuration the keyword *port* indicates the configuration is for port-based thresholds. This design decision was to allow for flow-based thresholds in the future. The next level down is the datapath ID — this is shown as the decimal value 153 or hex value 0xFF and was arbitrarily selected. Nested further down is the switch port number. Next is the metric name, which can be the name of any metric that is returned by the `ofp_port_stats_reply` message. Within each metric is the threshold value, which is compared against the current value, and optionally there is a derivative attribute that is used to define the desired order of derivative — or set to 0 to use the raw value.

```
port:
    153:
        2:
```

```
rx_bytes:
    threshold: 600000000
    derivative: 1
```

When new port statistics are received, processing of the thresholds is triggered. Each of the thresholds defined in the configuration file is compared to its associated current value. If the current value is greater than the threshold value then it is stored in a multi-dimensional `list` with the boolean value `True`. Otherwise, it is stored with the boolean value `False`. The list is then made available to the above Mirror layer. A potential improvement would be to read the thresholds from the configuration file at regular intervals rather than only when the application starts. This would allow adjustments to the thresholds without restarting the OpenFlow application.

4.2.3 Mirror layer

The mirror layer is responsible for maintaining a list of ports that need to be mirrored and a list of ports that are currently being mirrored. The first list is based on the data from the Tripwire layer. If any of the metrics for a port have their boolean value set to `True` then a threshold has been tripped on that port and therefore traffic ingress to that port must be mirrored. The Mirror layer sends a Flow-Mod message that instructs the switch to add a flow entry to the table with an ID of 1. The entry matches any traffic ingress to the port with the tripped threshold and the actions are to forward the packet on the port connected to the IDS and to *Goto-Table* 2. The purpose of the first action is to allow the IDS to inspect the traffic from the threshold-tripping port in greater depth. The second action allows the switch to perform regular packet forwarding functions for traffic entering that port. One of the goals of the project is to mitigate attacks with minimal disruption to legitimate traffic: until the IDS flags traffic as attack traffic the switch must assume that the traffic is legitimate and allow it to be processed in a regular manner.

Port mirroring for a port is disabled for a port when all values for that port fall below their associated thresholds. If there is a single attack flow amongst the traffic on the mirrored port then the IDS will generate an alert for that flow and the Firewall layer will instruct the switch to drop the attack flow traffic. Maintaining the assumption that there is only one attack flow, the process of dropping all traffic in that flow will most likely cause the metric values to fall back below their associated thresholds triggering the port mirroring to stop for that port. This is an ideal situation because the attack is mitigated and the IDS is not being made to filter through legitimate traffic. The other scenario is when there are multiple attack flows ingress to a port: if the IDS alerts on one of the flows and subsequent traffic for that flow is dropped then it is likely that the combined effort of all the attack flows will keep the current metric values above the threshold value. This will have the effect of continuing to port mirror traffic to the IDS which will alert on the remaining flows until enough flows are blocked to drive the metric values back below the thresholds.

4.2.4 Firewall layer

This is the component of Morepork that is made aware of attacks that have been confirmed to be present in all of the traffic that is passing through the switch. As mentioned earlier in this chapter, this layer is largely independent of all of the other layers. The only input of this layer is the stream of alerts from the IDS. The only output of this layer is a set of Flow-Mod messages sent to the switch. Alerts from the IDS are received by this layer through a UDP socket listening on port 514. The alerts take the form of *syslog* events. A regular expression is used to extract the source and destination IP addresses, the IP protocol (TCP or UDP), and

the source and destination port numbers. These fields are used to construct an `OFP_MATCH`, which is sent in a Flow-Mod command to the relevant switch. The flow entry is stored in table ID 0 and therefore packets are matched by the switch against this table before any other table. If an attack flow is matched in the table then action taken for packets in that flow is to drop them. The act in OpenFlow of dropping packets is achieved by sending the `OFPIT_APPLY_ACTIONS` instruction with an empty set for the action list [29]. On the other hand, the default action for packets that do not match a flow defined in table 0 is to attempt to match packets against table 1.

4.2.5 Main layer

Although this layer is described as the main layer it does not provide any of the core functionality of the application. Its primary purpose is to wrap all of the functionality from the other layers into an easy to use entry point that can be used to start the application. This layer also receives flow statistics and saves them to a Comma-Separated Value (CSV) file for additional performance analysis later.

4.3 Intrusion Detection System

Due to time constraints on the project it was desirable to find a zero-cost IDS solution that could be quickly integrated with the core project implementation.

4.3.1 Security Onion

At first, the virtual appliance *Security Onion* [3] was selected as the IDS of choice for this project because it is distributed as a live DVD ISO image that can be started by mounting the image in a virtual or physical machine and restarting the machine. It contains a collection of integrated intrusion detection tools including Snort for rule-based Network Intrusion Detection System (NIDS), Bro for analysis-based NIDS, and OSSEC for host-based intrusion detection. Analysis tools are also provided in the appliance including Sguil for drilling down, or *pivoting*, on the alerts from the IDSs, Squert as web-based interface for Sguil, Snorby for specifically analysing Snort alerts, and ELSA for log storage and searching. The appliance is based on Ubuntu Linux 12.04 thus providing a familiar environment for experienced Ubuntu Linux users. Upon starting the appliance the user can run the setup wizard which is located on the desktop once it has loaded. The setup wizard starts by asking the user to specify and configure the monitoring interface and the management interface. Then the user is offered the choice the simple approach, which is mostly unattended, or the advanced approach where the user can configure more specific details of each program in the appliance.

The Snort IDS provides a `local.rules` file that can be populated with custom rules. This a desirable feature for testing the alert system of Snort and to test the project implementation itself. The following snippet is a custom rule that was used to test the project implementation. The rule states that an alert should be generated when traffic is received from anywhere on any port and destined for anywhere on port 53. This means that any DNS traffic present on the mirror port with cause Snort to generate an alert. After adding the custom rule to `/etc/nsm/rules/local.rules` the rule is loaded into Snort by typing `sudo rule-update` in a terminal of the appliance.

```
alert udp any any -> any 53 (msg: "DNS amplification
    attack"; sid:9000700; rev:1)
```

Snort alerts are collected by Sguil along with alerts from the other IDSs present on the appliance. Sguil then records each alert in its log file */var/log/nsm/securityonion/sguild.log*. In order to send the alerts from the Security Onion appliance to the Morepork application, *syslog* is configured in the Security Onion appliance to scan the aforementioned log file for events starting with the phrase *sguil_alert* and forward the events to what it thinks is a syslog server on the machine running Morepork. The syslog configuration is defined in */etc/syslog-ng/syslog-ng.conf* and is initially activated by executing `sudo service syslog-ng restart`. The syslog events are sent in plain text to port 514 of the host running Morepork. An example of a raw syslog event is shown below with line breaks artificially added.

```
<13>Oct 26 08:32:44 489-securityonion sguil_alert:
    08:32:43 pid(3022)  Alert Received: 0 3 misc-activity
    489-securityonion-eth0 {2014-10-26 08:32:43} 6 38 {URL
    clients1.google.com} 192.168.10.128 74.125.19.113
    6 1299 80 10001 420042 1 38 38
```

The `SecurityOnion` class in the Morepork application acts as a syslog server and listens for syslog event messages. The event message is then parsed using a compiled regular expression and outputted as a `Dictionary` of fields relating to the alert. The `Dictionary` object is then passed to the `FirewallLayer` class with the aid of the `zope.event` package, which provides an observer pattern.

4.3.2 IDS Python script

Unfortunately, when the implementation was tested the Snort IDS did not always alert when attack packets were present in the mirrored traffic. Reasons for Snort failing are discussed in more depth in section 5.3. One of the goals of the project is for the IDS to be independent of the implementation and therefore the function of the IDS is beyond the scope of the project: the function only needs to receive mirrored traffic as an input and produce alerts when attack traffic is present as an output. Therefore a secondary IDS was created from scratch in a Python file. called *ids.py* (see appendix D.2). The Python script starts *tcpdump* as a subprocess and pipes the standard output of tcpdump to a buffer within the Python script. Whenever a packet is received on the monitoring interface of the IDS machine tcpdump outputs information about the packet to the standard output, including source and destination addresses. The Python script uses a compiled regular expression to extract the addresses and information about the packet. The information is reformatted to look like a Sguil alert and appended to */var/log/nsm/securityonion/sguild.log*. Since the output of the Python script intentionally looks the same as the output of Snort and Sguil, the syslog configuration remains the same.

4.4 Flow tables

Each switch that supports OpenFlow has a set of flow tables that are used to control flows of traffic through the switch [29]. Each entry in a flow table has a set of matches and a set of actions. The matches refer to specific attributes in the headers of a packet, although not all of the possible match fields need to be specified. Three common actions for a flow entry are forwarding the packet on one or more ports, sending the packet to the controller, or dropping the packet. Another possible action is `Goto-Table`, which sends the packet to the specified table for additional processing. The first flow table to process a packet has a table ID of *0*. If a Goto-Table command is issued then the packet must be sent to table with a higher ID number than the ID number of the current table. A default entry for a table, which

is known as the *Table-Miss* entry, is defined with an empty match set and an action that is usually either to goto another table or send the packet to the controller.

Figure 4.1 shows an example of the flow table entries that are created in the implementation. An incoming packet is first sent to table 0, which contains flow entries with no actions. This has the effect of dropping the packet if the packet matches that particular flow entry. The Table-Miss entry sends the packet to table 1 for processing. Table 1 contains instructions to forward packets on the port that has been designated for port mirroring. In the implementation port 3 was chosen because ports 1 and 2 were used for the Mininet hosts that send and receive the attack traffic. In addition to forwarding the packet to the mirror port, a matching packet is also sent to table 2 for processing. The extra action has the effect of allowing the packet to be delivered to its intended destination, thus minimising disruption to legitimate traffic. If the packet does not match a flow entry in table 1 then the default action is to send the packet to table 2. Table 2 matches on destination Ethernet addresses and forwards packets to the correct port — this is the standard function of a layer-two switch. See appendix C for a dump of the flow table from the Open vSwitch instance during one of the implementation tests.

table_id	**in_port**	**eth_dst**	**ip_src**	**ip_dst**	**udp_sport**	**udp_dport**	**Instructions**
0	1	aa:bb:cc:dd:ee:ff	1.2.3.4	5.6.7.8	12345	53	Apply-Actions {}
0	*	*	*	*	*	*	Goto-Table 1
1	1	*	*	*	*	*	Apply-Actions {Output: Goto-Table 2
1	*	*	*	*	*	*	Goto-Table 2
2	1	aa:bb:cc:dd:ee:ff	*	*	*	*	Apply-Actions {Output:
2	*	*	*	*	*	*	Packet-In

Table 4.1: Example flow table entries

Chapter 5

Evaluation

This chapter describes the environment used the test the implementation, the testing process, and the scenarios that were simulated during the tests. Finally, this chapters discusses the limitations of the test environment, and summarises the results of each test scenario.

5.1 Introduction

The test environment for this project consisted of two virtual appliances running on VirtualBox [27] virtual machines. The first appliance was a Mininet [17] appliance, which contains Mininet, Open vSwitch [26], and Ryu [38]. The second appliance was a Security Onion appliance, which contains a collection of tools including and augmenting intrusion detection systems (IDS). The tools are discussed in more depth in section 4.3.1. As discussed in section 4.3.1 the first version of the test environment utilised Snort IDS and Sguil to generate alerts in the presence of attack traffic. However, due to limitations that will be discussed later in section 5.3, Snort and Sguil were replaced with a Python script that generates alerts when any packets are received and logs the alerts to the same file that Sguil logs alerts to.

To test the implementation a network was needed that contained two standard hosts, a switch that supports OpenFlow, a host to run the OpenFlow controller, and a host to run the IDS. Mininet is a program that can simulate all of the aforementioned components except for the IDS, including the network in between [17]. A virtual network can be quickly created either through the Mininet command-line interface (CLI) or though Python scripts using the API. Initially the project relied on a Python script to define a simple network topology for Mininet but was later extended to provide complete test automation including the simulation of attack traffic. The final automation script is shown in appendix E.1. Mininet virtualises the switch component of the network using Open vSwitch [26], a virtual switch implementation that supports OpenFlow. This makes Mininet and Open vSwitch an ideal combination for testing a project that involves software-defined networking.

The Mininet topology definition is relatively straight forward and has been adopted from the Vandervecken implementation by Ewen McNeill. The Vandervecken topology consists of two hosts connected together with a single switch. This was adapted to include a bridge from the switch to two physical interfaces: the first bridge is used to send traffic to the IDS which runs on another virtual machine (VM), and the second bridge is used to send attack traffic from an another virtual machine if necessary. The connections between the VMs are provided by separate VirtualBox Internal Adapters, which means that traffic does pass through the networking stack of the host machine, and theoretically improves the performance of the connections. The later versions of the topology definition script also include commands to run the attack traffic simulations. Mininet provides a fully-scriptable interface

to the internals of Mininet and also allows some Linux commands to be executed on the virtual hosts that it creates. The virtual host receiving the attack traffic runs an iperf server that listens on UDP port 53. Iperf is a common bandwidth testing tool [21]. The virtual host sending the attack traffic runs the iperf client and sends a stream of packets to the server as fast as it can. UDP port 53 was selected to simulate DNS traffic. Before and after the attack traffic is send across the virtual network there are 30 second delays where the network is running but there is no network activity. This allows the tripwire layer to calibrate and creates a more distinguishable pattern when presenting the test results. In addition to running the test itself, the topology script starts another thread that dumps *per-port* statistics once every second. The data is exported to a CSV file for later analysis. Although the same statistics are used by the tripwire mechanism, the tripwire mechanism only requests statistics every 10 seconds which would reduce the sampling rate. Furthermore, the sampling rate of a second for the data export function was set to one second because any faster would potentially cause performance issues for the host machine running the tests and any slower would reduce the accuracy of the results.

5.2 Test scenarios

Each of the following test scenarios was executed using the automated test script described in the previous section. To automatically run and re-run the tests a wrapper script was created called `mn-run.sh` (shown in appendix E.2). The wrapper script calls `mn -c` which cleans up the Mininet environment and removes all of the virtual network interfaces that Mininet creates. Then Mininet is executed with the automation script as a parameter, which runs the full test. The wrapper script performs these two tasks in a loop, 50 times, to produce statistically integral results. Since the test environment is mostly automated, all of the test parameters remain exactly the same — except for the parameters that are being tested. Therefore, most of the test results follow a similar pattern.

As shown in appendix E.1 the Comma-Separated Value (CSV) file containing port statistics that is created with each test execution has a UNIX time stamp in the file name to avoid file name collision with different runs of the test. Each line of each CSV file originally included a UNIX time stamp value for unique, incrementing x-axis values however, aggregating the statistics from each run of a test scenario became too difficult and the time stamp value was replaced with a duration value, in seconds, that is calculated relative to the start time of the test run.

Tableau Desktop (TD) [40] was used to aggregate the data from each test run and to create a visualisation. The CSV files have separated values for the number of transmitted and received bytes on each port. TD does not allow multiple measures to be presented on the same graph; instead drawing them on separate graphs one below another. This was a problem because the visualisation needed to compare the number of received bytes on the ingress port against the number of bytes transmitted on the egress port. Since the environment was completely simulated it was assumed that there would be no traffic flowing in the opposite direction. Therefore, a *Calculated Measure* was created in TD to combine the number of received and transmitted bytes on each port. It was also not necessary to present the measures for all four ports in each test scenario so filtering was applied in TD to display only the relevant ports for each test scenario. The measures of transmitted and received bytes provided by Open vSwitch instances are cumulative, so the data was transformed in TD to present the difference in cumulation (also known as rate or bytes per second).

5.2.1 High bandwidth flow with attack traffic

The first test scenario tests the base case for this project: attack traffic is sent through a switch running the implementation and the attack traffic is detected and mitigated. Although this behaviour does not reflect a real-world a network environment, it was desirable to build the test environment where only a single flow of traffic was for the simulation. Such an environment increases the likelihood that clear-set results demonstrating whether the implementations works or does not work will be available. Subsequent test scenarios are provided to simulate other flows of traffic in a controlled manner.

Figure 5.1 shows the attack traffic is starting to enter the switch at 30 seconds into the test and immediately exit the switch. Then, 10 seconds later when the port statistics become available to the tripwire system the threshold is flagged as having been reached and the traffic is mirrored to the mirror port which feeds into the IDS. Within one the IDS generates an alert for the attack traffic and notifies the firewall layer of the implementation. The firewall layer inserts flow entry to drop the attack traffic and one second later the attack traffic is dropped rather than exiting the switch. The attack traffic continues to be sent to the switch until 150 seconds into the test where it stops and within fours seconds there is no longer any traffic flowing though the virtual network at all.

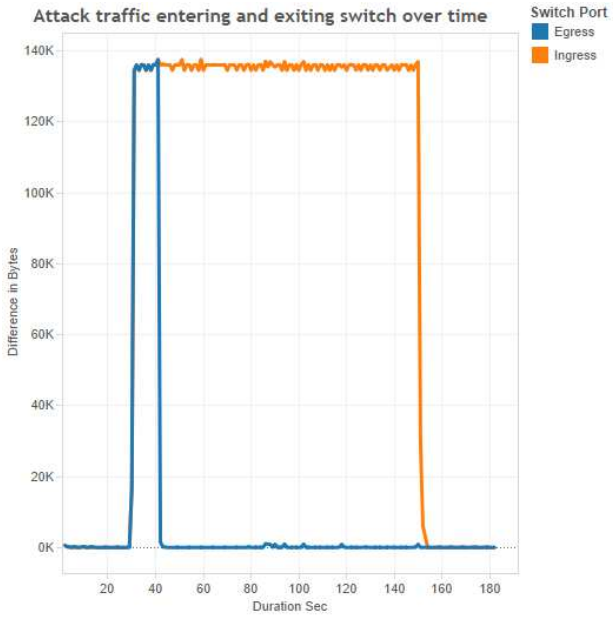

Figure 5.1: Results of simulating high-bandwidth attack flow across the implementation

5.2.2 High bandwidth flow with legitimate traffic

To test that legitimate traffic is unaffected by the implementation a scenario has been created where a single flow of legitimate passes through the switch with the implementation run-

ning. The legitimate traffic is a high-bandwidth flow and therefore will exceed the threshold defined in the tripwire system. An example of a real-life scenario where this test would be applicable would be a switch on the edge of a relatively small network: the network would have a relatively low threshold and therefore a file transfer, for example, would consume a large amount of the available bandwidth but can be considered legitimate traffic.

The legitimate traffic, as shown in figure 5.2 begins to enter the switch at 30 seconds into the test and immediately exits the switch. 10 seconds later, when the port statistics are collected by the tripwire system, the flow is treated as suspicious and the ingress port traffic is mirrored to the IDS (shown in figure 5.3). Since the IDS does not detected any attack traffic it does not generate an alert. Therefore, the legitimate traffic continues to exit the switch until iperf stops sending traffic at 150 seconds into the test.

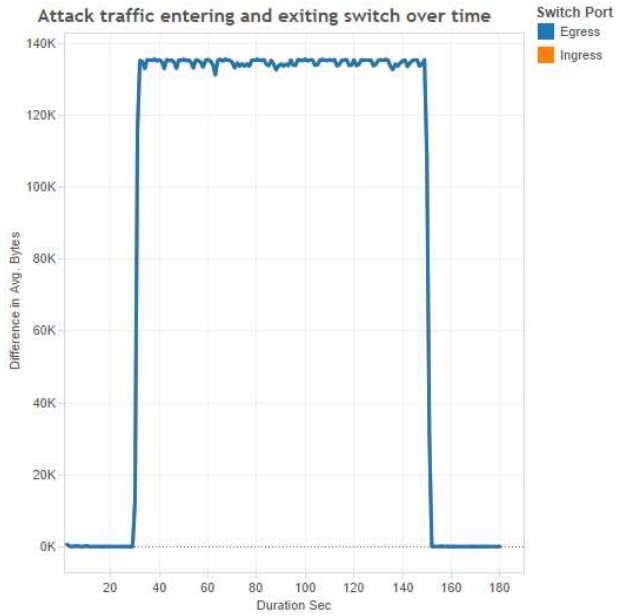

Figure 5.2: Results of simulating a legitimate high-bandwidth flow across the implementation

An improved version of this scenario would include a flow of legitimate traffic and a flow of attack traffic entering a switch on the same port. The combination of the two flows would exceed the threshold defined in the tripwire system and both flows would be mirrored to the IDS. The intended outcome would be an alert from the IDS matching the flow of the attack traffic which would result in a flow entry to drops packets from the attack flow — the legitimate traffic flow would not match this flow entry and would continue to be forwarded by the switch as per normal. Due to the limitation in `ids.py` that *all* traffic received on the mirror port is flagged as attack traffic it was not possible to build this enhanced scenario within the time constraints.

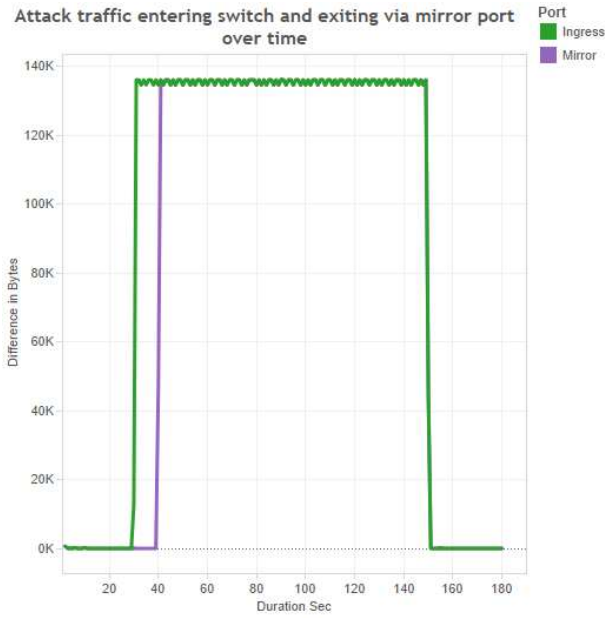

Figure 5.3: Results demonstrating the port traffic being mirrored to the IDS

5.2.3 Low bandwidth request traffic with high bandwidth response traffic

While the scenario in section 5.2.1 tests for any kind of high-bandwidth attack, it does not necessarily test the specific vectors of a DNS amplification attack. Like other amplification attacks, a DNS amplification attack sends a small request packet with a forged source address and the unsuspecting server replies with a large response that goes to the forged address rather than to the attacker. Figure 5.4 shows an example of a network topology where the attack sends spoofed traffic from its own network to a network with an open DNS resolver and the replies are sent to a victim host in a third network. The edge routers between these networks are connected via switches (for example, in an internet exchange) and both the switches in this scenario are running the implementation. The hypothesis of this scenario is that the switch between the attacker's network and the network with the open DNS resolver will not detect the attack because the request traffic will not consume enough bandwidth to reach the tripwire threshold. However, the response traffic will be large enough to reach the threshold and therefore the switch between the network with the open DNS resolver and the victim network will detect and mitigate the attack.

The attack starts at 30 seconds into each test run with the request traffic entering and exiting the first switch without triggering the port mirroring (see figure 5.5). At the same time the response traffic begins entering the second switch and exiting towards the victim network. Another 10 seconds pass and the controller application receives port statistics from both switches. Since the traffic passing through the first switch does not reach the threshold value port mirroring is not enabled for the ingress port of the first switch. Therefore, the

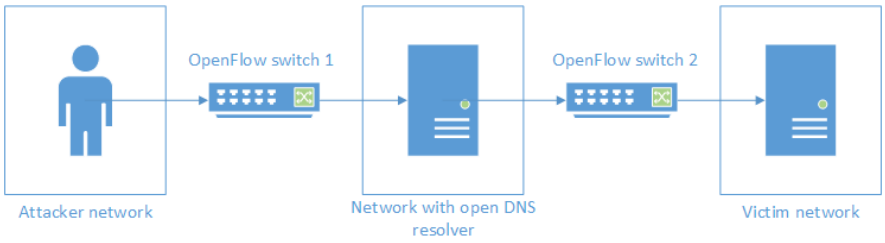

Figure 5.4: Network topology diagram with two switches

request traffic in the attack is treated as legitimate traffic and the traffic flow follows the pattern for a general flow of legitimate traffic as demonstrated in section 5.2.2. On the other hand, the traffic flowing through the second switch consumes enough bandwidth to trigger port mirroring. Therefore, the response traffic in the attack is treated as legitimate traffic and the traffic flow follows the pattern for a general flow of attack traffic as demonstrated in section 5.2.1.

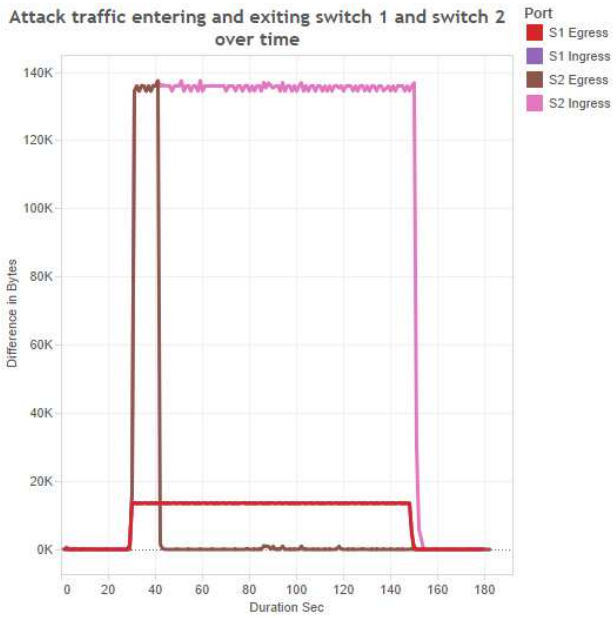

Figure 5.5: Results of simulating a DNS amplification attack with two switches

5.3 Limitations of the test environment

A major limitation of the test environment is the CPU of the virtual machine host. The host CPU is an Intel Core i5-750 running at 2.66GHz and has four cores that can execute four threads in total at once. Both the Mininet virtual machine (VM) and Security Onion VM were allocated a single core and four gigabytes of RAM each. When a test run was being executed a ELSA was consuming 100% of the CPU allocated to the Security Onion VM. Despite Snort being disabled when `ids.py` was used, another tool on the Security Onion VM was collecting full packet captures and storing them on disk. ELSA attempted to index these full packet captures and hence consumed all of the CPU available to that VM. On the Mininet VM, Iperf consumed a lot of CPU cycles in order to generate the large stream of packets. Open vSwitch also consumed a lot of CPU cycles in order to process and forward the packets. The contention for the CPU on the Mininet VM seemed to directly impact the throughput during the attack tests. Due to this limitation it was not possible to test the goal of running the implementation with an amount of traffic that would commonly pass through an internet exchange. This also affected the rate at which requests for port statistics were responded to and received by the implementation. Requests for ports statistics are send by the implementation every 10 seconds. Reducing this interval caused replies to be received at inconsistent intervals — multiple replies were often received at once. Increasing the interval reduced the accuracy of the implementation. It is possible that increasing the number of cores available to the Mininet VM would have improved the performance of the test environment. Alternatively, the Iperf server and client could have been executed on separate VMs, reducing contention for the CPU within the Mininet VM, but could have increased contention for the CPU on the virtual machine host.

Another major limitation of the test environment was the unreliability of the IDS: occasionally it would not generate alerts when attack traffic was mirrored to the Security Onion VM. As mentioned earlier, a tool on the Security Onion VM stores every packet that is received on disk for future forensic analysis. With the large number of test iterations and the small disk attached to the VM, the disk eventually ran out of disk space and the operating system did not provide any indication that there was no remaining disk space. Sguil normally logs alerts to a file on disk, but when the disk ran out of space this was no longer possible. Since the log was not being updated syslog had no data to send back to the implementation. The web interfaces for the tools provided by Security Onion rely on free disk space to store temporary files when the applications are responding to web requests. Since there was no disk space the web interfaces failed to load — making it more difficult to determine the cause of the incident. Once identified, the solution was to create a new VM running Security Onion with a much larger disk.

Since it was not immediately apparent why the IDS was not alerting a simple Python script was created to simulate the combined function of the Snort IDS and Sguil. The script is called `ids.py` and is discussed in more detail in section 4.3.2. The script was executed from within the Security Onion VM (with Snort and Sguil stopped) and it seemed to work when the test environment was operated manually. However, when the automation script was used `ids.py` also did not seem to work. The problem with Snort was assumed to be a problem with the attack traffic not being matched by the attack definition, however `ids.py` was designed to generate alerts for all traffic received on the monitoring interface to avoid this issue. It became apparent that `ids.py`, or tcpdump when executed directly, were not receiving traffic from the monitoring interface at all — although in some situations traffic would come through. VirtualBox [27], the virtualisation tool used on the host machine, has an option to disable or enable promiscuous mode for the host network interface or for VM network interfaces. However, the option was set to enable promiscuous mode in all circumstances.

Assuming this was the problem the Security Onion VM would not have been allowed to enable promiscuous mode which would have prevented all of the packet capturing tools from capturing packets. However, this did not seem to be the problem either because the network activity indicator for the Security Onion VM showed that the traffic was not being received by the VM at all, however the traffic was being transmitted by the Mininet VM. Another symptom of the issue as experienced by `ids.py` and tcpdump occurred whenever the automated test script was executed: the packet capture would stop prematurely. The most significant difference between the automated execution of a test and the manual execution of a test was the order in which components were started: in the automated test `ids.py` was started first and the automated script was started second. In a manual test Mininet was started first, then `ids.py`, and finally iperf for the attack simulation. When Mininet is started it instructions Open vSwitch to create virtual network interfaces and attach itself to the VM's physical network interfaces (in the case of this project's network topology definition). Although not confirmed, it is suspected that Open vSwitch disables the network interface momentarily in order to attach itself to the interface. When the interface is disabled it may disrupt the interface in the Security Onion VM since the interfaces of the two VMs are connected together with VirtualBox internal network adapters. This issue was unresolvable, and the workaround during the tests was to manually start and stop `ids.py` during the warm-up and cool-down phases of each test run.

5.4 Summary

To evaluate the implementation three test scenarios were constructed and executed in an controlled environment using Mininet. The first test scenario sent a single flow of attack traffic through a switch running the implementation with the expectation that the flow would be stopped. This was to test the implementation against the objective, which is to detect and to mitigate DNS amplification attacks. Since the parameters of the test scenario are quite vague, the scenario also tests the goal of being extensible so that other attack types can be detected and mitigated. The results suggest that after a small window where the attack successfully passes through the switch the attack flow is dropped. The second test scenario sent a single flow of legitimate traffic through a switch running the implementation with the expectation that the flow would not be stopped. This was to test the requirement that the implementation has little impact on legitimate traffic. The results suggest that the legitimate traffic is completely unaffected by the implementation despite the traffic exceeding the threshold and being deemed as suspicious. The third test scenario sent a small amount of attack through one switch and a large amount of attack traffic through another switch to simulate an amplification attack where requests are small and responses are large. The purpose of this test was to test the objective specifically, rather than in the broad sense. The results suggest that a DNS amplification attack can be mitigated when the DNS responses pass through a switch running the implementation, but it is possible that the DNS requests will not detected or mitigated by the implementation. The results also suggest that the implementation would be more effective with more locations on the internet running the implementation, however the solution is still likely to detect and mitigate attacks with a small number of instances of the implementation running.

Chapter 6

Conclusion and future work

DNS amplification attacks are currently a common technique used by attackers to deny service to legitimate users. This has become a problem because attackers able to construct these attacks with large amplification factors that cause majors amounts of congestion and previously suggested techniques to mitigate these attacks have been unsuccessful. While the project implementation is successful in terms of the objective, there are some weaknesses to implementation too. This chapter seeks to discuss the strengths and weaknesses of the project, and to suggest future work that could improve this project.

6.1 Strengths of the implementation

Section 5.4 made the suggestion that the implementation has met the objective of the course. This section will discuss some of the specific project requirements that have also been achieved.

The implementation is supposed to run on a variety of hardware or software. Due to time constraints the implementation was not tested on hardware. However, the implementation supports any device that supports OpenFlow version 1.3 and therefore the onus is on switch vendors to provide hardware or software switches. The core implementation is a software application that runs on top of Ryu — a software-based OpenFlow controller. As demonstrated with the test environment, Morepork and Ryu can run in a virtual machine if necessary, but in theory could run on a physical machine too. The IDS has been treated as a black box that accepts a specific input and produces a specific output. There is no expectation as to how the Intrusion Detection System (IDS) is implemented which means it could also run in hardware or software.

The black box approach of the IDS also addresses the extensibility of the project. The attacks that can be detected by the implementation depends directly on the configuration and implementation of the IDS. There are large rule sets available for Snort that could be used with the project. Additionally, Snort allows for custom rules to be defined — which was the preferred method for evaluating this project.

The implementation reduces the impact of the detection and mitigation mechanism on legitimate traffic by offloading the responsibility of detecting attacks to another machine (the IDS). If the IDS fails for some reason or the link between the switch and the IDS becomes congested or otherwise unavailable then the switch will stop mitigating attacks but will continue to function as a normal switch.

The implementation is designed to run on a switch, not a router, and therefore addresses this requirement slightly differently. The third test scenario (see section 5.2.3) demonstrates that although it is not necessary to run the implementation on all switches it would definitely improve the effectiveness of the implementation. A more important factor is the run the

implementation on switches that handle large amounts of internet traffic and are potentially subject to the presence of more attack traffic. To address this factor the goal of running the implementation in an internet exchange environment (or with an equivalent amount of traffic) was introduced to the project.

6.2 Weaknesses of the implementation

Testing with the amount of traffic commonly seen in an internet exchange (IX) was not possible due to the limitations of the test environment. Therefore, there is no evidence that the implementation could operate in an IX. On the other hand, future work could be done to improve the test environment and this may be enough to demonstrate the implementation working with an exchange-level amount of traffic.

Since the objective of the project is targeted at DNS amplification attacks and more generally distributed denial of service attacks (DDoS) there is an assumption that the attacks being mitigated will have large flows of attack traffic. For small-scale DDoS attacks or for attacks that rely on vectors that do not include sending large amounts of network traffic, this implementation is suggested to be ineffective. Furthermore, the implementation assumes the switch is trying to protect its own amount of available bandwidth rather than stopping the attack itself. If the threshold on a port is set to a value that is slightly higher than the daily peak load value for that port and the attacker decides to execute their attack during an off-peak time then the implementation will not detect or mitigate the attack. On the other hand, an attack during the off-peak time of both the switch and the victim of the attacker is less likely to cause denial of service and by definitely of *peak* is going to impact fewer legitimate users.

6.3 Future work

This section discusses potential changes and improvements to the project for the future.

For reasons discussed in section 3.1.2 it is desirable to reduce the amount of traffic that is mirrored to the IDS. The current project design reduces the traffic by deciding on a per-switch port basis whether the traffic needs to be mirrored or not. It is unlikely that all of the traffic entering a port will be attack traffic and therefore the solution could be improved by mirroring traffic on a per-flow basis. The port-based mirroring would provide a coarse-gained filter in attempt to find an attack flow and once a port has been flagged as suspicious then the mirroring process could be applied per-flow for that specific port. Traffic would only be mirrored once one or more traffic flows have been flagged as especially suspicious. On the other hand, thresholds would be unsuitable to whether a particular flow is suspicious. It is unlikely that each flow has the same characteristics, which would require individual thresholds for individual flows and the thresholds would need to change. If this process were to be applied then machine-learning would be required to improve the accuracy of the detection process.

Once an attack has been detected by the IDS and the implementation has blocked the attack flow, then assuming that no other attack traffic has been detected from the port being mirrored then the port should stop being mirrored. This another attempt to reduce the amount of traffic being sent to the IDS. Furthermore, it is possible that multiple ports trip their thresholds at once and the implementation will mirror traffic from those ports to the IDS. If the combined bandwidth of the ports being mirrored exceeds the capacity of the link between the switch and the IDS then some of the mirrored traffic may be dropped. Although this will not impact the forwarding of legitimate packets through the switch it will impact

the ability for the IDS to detect attack traffic. If the IDS becomes overwhelmed with traffic then it may not be able to alert the switch about specific attacks. On the other hand, if it does manage to inform the switch about an attack then the switch can block the attack flow which will stop that flow from being mirrored to the IDS. This behaviour is possible because the flow table with the drop rules are evaluated before the mirror rules and forwarding rules - this was an intentional part of the design. To ensure the IDS is never overwhelmed with traffic the implementation could set a limit on the number of ports that can have traffic mirrored to the IDS. Combining with the idea from above, when no attacks are detected by the IDS in the mirrored traffic then those ports should stop being mirrored to the IDS and the other ports can then be mirrored.

Due to time constraints the code to remove an attack flow entry from the flow table was never implemented. Although this does not immediately cause any immediate issues for any of the components it may cause issues for legitimate traffic in the future that coincidently matches the attack flow vectors (specifically for DNS requests or responses). Also, some of the configuration was hard-coded within the code of each layer of the implementation — for example the port number that traffic is mirrored to. These configuration options should be moved to an external configuration file and the code to read the configuration file should be moved from the tripwire layer to a layer that allows all the other layers to access the configuration.

Appendix A

Example of a large DNS response

This is an example of a large DNS response that has been generated by the command-line utility *dig*. The lines are truncated, however the message size is recoreded as 8320 bytes at the bottom of the output — this demonstrates the full size of the response.

```
$ dig ANY doc.gov @x.x.y.y
;; Truncated, retrying in TCP mode.

; <<>> DiG 9.8.3-P1 <<>> ANY doc.gov
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60091
;; flags: qr rd ra; QUERY: 1, ANSWER: 44, AUTHORITY: 6, ADDITIONAL: 4

;; QUESTION SECTION:
;doc.gov.                       IN   ANY

;; ANSWER SECTION:
doc.gov.            10798   IN   RRSIG   SOA 8 2 10800 20141119070630 20141020060630
doc.gov.            10798   IN   RRSIG   SOA 8 2 10800 20141119070630 20141020060630
doc.gov.            10798   IN   SOA hchbens1.doc.gov. dnsadmin.doc.gov. 2014072865 3
doc.gov.            10798   IN   RRSIG   TXT 8 2 10800 20141106123148 20141007123106
doc.gov.            10798   IN   RRSIG   TXT 8 2 10800 20141106123148 20141007123106
doc.gov.            10798   IN   TXT "MS=ms91611337"
doc.gov.            10798   IN   TXT "khbl4plXHgnKtHsVtFp6P2+2C/Tpn/Vl/4xMUy+3yN8KPPz
doc.gov.            10798   IN   RRSIG   MX 8 2 10800 20141114023329 20141015020647 1
doc.gov.            10798   IN   RRSIG   MX 8 2 10800 20141114023329 20141015020647 1
doc.gov.            10798   IN   MX  10 smtpedge1.uspto.gov.
doc.gov.            10798   IN   MX  10 smtpedge2.uspto.gov.
doc.gov.            298 IN   RRSIG   AAAA 8 2 300 20141107173951 20141008164211 11766
doc.gov.            298 IN   RRSIG   AAAA 8 2 300 20141107173951 20141008164211 14811
doc.gov.            298 IN   AAAA    2610:20::20:5ec:d0c:d0c:d0c
doc.gov.            298 IN   RRSIG   A 8 2 300 20141113001227 20141013234923 11766 do
doc.gov.            298 IN   RRSIG   A 8 2 300 20141113001227 20141013234923 14811 do
doc.gov.            298 IN   A   170.110.225.194
doc.gov.            3598    IN   RRSIG   NSEC 8 2 3600 20141108231129 20141009230013
doc.gov.            3598    IN   RRSIG   NSEC 8 2 3600 20141108231129 20141009230013
doc.gov.            3598    IN   NSEC    associate.doc.gov. A NS SOA MX TXT AAAA RRSI
```

```
doc.gov.          298 IN  RRSIG   DNSKEY 8 2 300 20141116000000 20141016230000 11209
doc.gov.          298 IN  RRSIG   DNSKEY 8 2 300 20141116000000 20141016230000 11766
doc.gov.          298 IN  RRSIG   DNSKEY 8 2 300 20141116000000 20141016230000 42040
doc.gov.          298 IN  RRSIG   DNSKEY 8 2 300 20141116000000 20141016230000 51528
doc.gov.          298 IN  DNSKEY  256 3 8 AwEAAdoSoUIQleHGNf1pPVyUv3fegOEqKQDur1dquT
doc.gov.          298 IN  DNSKEY  256 3 8 AwEAAdtKPEiB4OqKghepiGwUBqkdbRYk8sg8UBEXp
doc.gov.          298 IN  DNSKEY  256 3 8 AwEAAeJRixTIyyo0QVfhsDtHNneV1rOmrCkHOwi90h
doc.gov.          298 IN  DNSKEY  256 3 8 AwEAAfctEhSjYDJW3Tq6cQv8z3uC+N+Q8pA9ozC6s
doc.gov.          298 IN  DNSKEY  257 3 8 AwEAAaMW+Kqwn3Zokh7OxWV6iQOI3HqWcaWnYYyLs+
doc.gov.          298 IN  DNSKEY  257 3 8 AwEAAcCAMAdCOvmCdsMBh8+5EqP4c0jTO86Ht0/IW9
doc.gov.          298 IN  DNSKEY  256 3 8 AwEAAbkCPpGWIZ0DYbxnYj06yHzrFzRnQH+RDhfHN
doc.gov.          10798   IN  RRSIG   NS 8 2 10800 20141114022009 20141015013431 117
doc.gov.          10798   IN  RRSIG   NS 8 2 10800 20141114022009 20141015013431 148
doc.gov.          3598    IN  RRSIG   DS 8 2 3600 20141027161018 20141020161018 1321
doc.gov.          3598    IN  DS  42040 8 1 4D8A621E8A7AC8AED30720D0C0F1B74CC250DF26
doc.gov.          3598    IN  DS  42040 8 2 41BA3D5F5B7BB2F9C599F2D21E1A3CE24B37DAB
doc.gov.          3598    IN  DS  11209 8 1 230E4D6480036A2432EA74AE0D3C5C3AE9341C29
doc.gov.          3598    IN  DS  11209 8 2 00A2783429DF6A9533B998C92C322FF3A9169287
doc.gov.          10798   IN  NS  dnssec7.datamtn.com.
doc.gov.          10798   IN  NS  dnssec10.datamtn.com.
doc.gov.          10798   IN  NS  hchbens1.doc.gov.
doc.gov.          10798   IN  NS  gpaens2.doc.gov.
doc.gov.          10798   IN  NS  dnssec11.datamtn.com.
doc.gov.          10798   IN  NS  dnssec9.datamtn.com.

;; AUTHORITY SECTION:
doc.gov.          10798   IN  NS  dnssec9.datamtn.com.
doc.gov.          10798   IN  NS  dnssec10.datamtn.com.
doc.gov.          10798   IN  NS  gpaens2.doc.gov.
doc.gov.          10798   IN  NS  dnssec11.datamtn.com.
doc.gov.          10798   IN  NS  dnssec7.datamtn.com.
doc.gov.          10798   IN  NS  hchbens1.doc.gov.

;; ADDITIONAL SECTION:
gpaens2.doc.gov.    86394   IN  A   170.110.225.13
gpaens2.doc.gov.    10793   IN  AAAA    2610:20::20:d0c:90c:225:223
hchbens1.doc.gov.   86394   IN  A   170.110.225.11
hchbens1.doc.gov.   10793   IN  AAAA    2610:20::20:d0c:90c:225:225

;; Query time: 797 msec
;; SERVER: x.x.y.y#53(x.x.y.y)
;; WHEN: Tue Oct 21 09:32:45 2014
;; MSG SIZE  rcvd: 8230
```

Appendix B

Morepork source code

Morepork is the name given to the project implementation. This is the primary contribution of the project. A description of the code is found in chapter 4. Note that the longer lines are truncated.

B.1 simple_monitor.py

```
from operator import attrgetter

from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import hub

class SimpleMonitor(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)

    @set_ev_cls(ofp_event.EventOFPStateChange,
        [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        '''
        Keep the list of datapaths up-to-date.
        '''
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if not datapath.id in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister datapath: %016x', datapath.id)
```

```
                del self.datapaths[datapath.id]

    def _monitor(self):
        '''
        Poll each datapath for new stats.
        '''
        while True:
            for dp in self.datapaths.values():
                self._request_stats(dp)
            hub.sleep(10)

    def _request_stats(self, datapath):
        self.logger.debug('send stats request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        req = parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)

        req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
        datapath.send_msg(req)

    @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def _port_stats_reply_handler(self, ev):
        body = ev.msg.body

        self.logger.info('datapath         port     '
                         'rx-pkts  rx-bytes rx-error '
                         'tx-pkts  tx-bytes tx-error')
        self.logger.info('---------------- -------- '
                         '-------- -------- -------- '
                         '-------- -------- --------')

        for stat in sorted(body, key=attrgetter('port_no')):
            self.logger.info('%016x %8x %8d %8d %8d %8d %8d %8d',
                ev.msg.datapath.id, stat.port_no,
                stat.rx_packets, stat.rx_bytes, stat.rx_errors,
                stat.tx_packets, stat.tx_bytes, stat.tx_errors)
```

B.2 layer_tripwire.py

```
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_DI
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub

from morepork.app import simple_monitor
from morepork.tripwire import collector
from morepork.tripwire import tripwire
```

```
class TripwireLayer(simple_monitor.SimpleMonitor):

    tripwire_table_id = 2
    poll_time = 10

    def __init__(self, *args, **kwargs):
        super(TripwireLayer, self).__init__(*args, **kwargs)
        self.collector = collector.PortStatsCollector()
        self.tripwire = tripwire.Tripwire(self.logger, self.collector)

    def add_flow(self, datapath, priority, match, actions):
        '''
        Override method in simple_switch_13 so we can place
        switch flows in the table number of our choosing.
        '''
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst, table_id=sel
        datapath.send_msg(mod)

    def add_default_flow(self, datapath, table_id, goto_table_id=None):
        '''
        Adds a default flow for a table. Default action is
        to goto the next table (specified by goto_table_id).
        '''
        if not goto_table_id: return

        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = parser.OFPMatch()

        inst = [parser.OFPInstructionGotoTable(goto_table_id)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=0,
            match=match, instructions=inst, table_id=table_id)

        datapath.send_msg(mod)

    def print_tripwires(self):
        wires = self.tripwire.process_port_stats()
        self.logger.info('datapath         port     '
                         'metric           value        '
                         'threshold      tripped')
        self.logger.info('---------------- -------- '
                         '---------------- ------------ '
                         '-------------- -------')
```

```
        for item in wires:
            self.logger.info('%016x %8x %16s %13d %13d %7s',
                item.dpid, item.port_no, item.metric, item.value,
                item.threshold, item.tripped)

        for item in wires:
            if item.tripped:
                self.logger.warning('Tripped: datapath=%x port=%x metric=%s %d
                    item.dpid, item.port_no, item.metric, item.value,
                item.threshold)

    def _monitor(self):
        '''
        Override method in simple_monitor so we can change
        how often we request stats.
        '''
        while True:
            for dp in self.datapaths.values():
                self._request_stats(dp)
            hub.sleep(self.poll_time)


    @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def _port_stats_reply_handler(self, ev):
        super(TripwireLayer, self)._port_stats_reply_handler(ev)
        dpid = ev.msg.datapath.id
        body = ev.msg.body

        self.collector.push(dpid, body)
        self.print_tripwires()
```

B.3 layer_mirror.py

```
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_D
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub

from morepork.app import layer_tripwire

class MirrorLayer(layer_tripwire.TripwireLayer):

    mirror_table_id = 1
    mirror_port = 3 # TODO make this configurable
    mirror_map = {}

    def __init__(self, *args, **kwargs):
        super(MirrorLayer, self).__init__(*args, **kwargs)

    def get_mirror_ports(self):
```

```
        return self.mirror_in_ports

    def add_mirror_port(self, dpid, in_port):
        '''
        Add action to mirror traffic to the mirror port,
        then go to the switch table to forward the traffic
        normally.
        '''
        datapath = self.datapaths[dpid]
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = parser.OFPMatch(in_port=in_port)

        actions = [parser.OFPActionOutput(self.mirror_port)]
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
            actions), parser.OFPInstructionGotoTable(self.tripwire_table_id)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=1,
            match=match, instructions=inst, table_id=self.mirror_table_id)
        datapath.send_msg(mod)

        self.logger.warning('Added port mirror %d --> %d on datapath %x',
            in_port, self.mirror_port, dpid)

    def remove_mirror_port(self, dpid, in_port):
        datapath = self.datapaths[dpid]
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        match = parser.OFPMatch(in_port=in_port)

        mod = parser.OFPFlowMod(datapath=datapath, match=match,
            command=ofproto.OFPFC_DELETE, table_id=self.mirror_table_id,
            out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY)
        datapath.send_msg(mod)

        self.logger.warning('Removed port mirror %d --> %d on datapath %x',
            in_port, self.mirror_port, dpid)

    def update_mirror_ports(self, wires):
        new_map = {}
        for wire in wires:
            if wire.dpid not in new_map:
                new_map[wire.dpid] = {}

            if wire.port_no not in new_map[wire.dpid]:
                new_map[wire.dpid][wire.port_no] = wire.tripped

            if (not new_map[wire.dpid][wire.port_no] and
                wire.tripped):
                new[wire.dpid][wire.port_no] = True
```

```
        old_map = self.mirror_map
        for dpid, ports in new_map.iteritems():
            for port_no, tripped in ports.iteritems():
                if dpid not in old_map:
                    old_map[dpid] = {}

                if port_no not in old_map[dpid]:
                    old_map[dpid][port_no] = False

                if tripped and not old_map[dpid][port_no]:
                    self.add_mirror_port(dpid, port_no)
                elif not tripped and old_map[dpid][port_no]:
                    self.remove_mirror_port(dpid, port_no)
                # else if new=true and old=true
                # => already port mirroring
                # else if new=false and old=false
                # => never tripped

        self.mirror_map = new_map

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        super(MirrorLayer, self).switch_features_handler(ev)

        datapath = ev.msg.datapath
        self.add_default_flow(datapath, self.mirror_table_id, self.tripwire_ta

    @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def _port_stats_reply_handler(self, ev):
        super(MirrorLayer, self)._port_stats_reply_handler(ev)

        wires = self.tripwire.process_port_stats()
        self.update_mirror_ports(wires)
```

B.4 layer_firewall.py

```
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_DI
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub
from ryu.ofproto import ether
from ryu.ofproto import inet

from morepork.app import layer_mirror
from morepork.ids import securityonion

class FirewallLayer(layer_mirror.MirrorLayer):

    firewall_table_id = 0
    firewall_drops = {}
```

```
ids_listen_address = '0.0.0.0'
ids_listen_port = 514

def __init__(self, *args, **kwargs):
    super(FirewallLayer, self).__init__(*args, **kwargs)
    self.ids = securityonion.SecurityOnion(self.logger)
    self.ids.ids_alert_subscribe(self.ids_alert_handler)
    self.ids.listen(self.ids_listen_address, self.ids_listen_port)

def get_firewall_drops(self):
    return self.firewall_drops

def add_firewall_drop(self, dpid, ev):
    '''
    If traffic flow should be dropped then add flow rule
    to match that flow and apply no actions. There is no
    goto table instruction to prevent the normal switch
    functions and thus dropping the packet.
    '''
    datapath = self.datapaths[dpid]
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    src_port = int(ev.src_port)
    dst_port = int(ev.dst_port)
    if ev.proto == '6': # TCP
        match = parser.OFPMatch(ipv4_src=ev.src_ip, ipv4_dst=ev.dst_ip,
            tcp_src=src_port, tcp_dst=dst_port, eth_type=ether.ETH_TYPE
            ip_proto=inet.IPPROTO_TCP)
        proto = 'TCP'
    elif ev.proto == '17': # UDP
        match = parser.OFPMatch(ipv4_src=ev.src_ip, ipv4_dst=ev.dst_ip,
            udp_src=src_port, udp_dst=dst_port, eth_type=ether.ETH_TYPE
            ip_proto=inet.IPPROTO_UDP)
        proto = 'UDP'
    else:
        return

    actions = []
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
        actions)]
    mod = parser.OFPFlowMod(datapath=datapath, priority=1,
        match=match, instructions=inst, table_id=self.firewall_table_id)
    datapath.send_msg(mod)

    self.logger.warning('Adding entry to drop traffic flow %s %s:%s -->
        proto, ev.src_ip, ev.src_port, ev.dst_ip, ev.dst_port, dpid)

def remove_firewall_drop(self, dpid, match):
    datapath = self.datapaths[dpid]
```

```
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        mod = parser.OFPFlowMod(datapath=datapath, match=match,
            command=ofproto.OFPFC_DELETE, table_id=self.firewall_table_id,
            out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY)
        datapath.send_msg(mod)

        self.logger.warning('Removed flow to drop traffic')

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        super(FirewallLayer, self).switch_features_handler(ev)

        datapath = ev.msg.datapath
        self.add_default_flow(datapath, self.firewall_table_id, self.mirror_ta

    @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def _flow_stats_reply_handler(self, ev):
        super(FirewallLayer, self)._flow_stats_reply_handler(ev)

        # Check flow stat threshold for drop flows
        # If stat < threshold
        #     remove_firewall_drop(...)

    def ids_alert_handler(self, ev):
        # Stop the attack on all datapaths
        for dpid in self.datapaths:
            self.add_firewall_drop(dpid, ev)
```

B.5 main.py

```
import datetime
import os.path

from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_D
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub

from morepork.app import layer_firewall

class MoreporkApp(layer_firewall.FirewallLayer):

    flow_stats_file = 'flow-stats-%s.csv'

    def __init__(self, *args, **kwargs):
        super(MoreporkApp, self).__init__(*args, **kwargs)

        t = datetime.datetime.today()
```

```
        self.flow_stats_file = self.flow_stats_file % t.strftime('%s')

        # TODO init config here and replace TRIPWIRE_CONF

    @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def _flow_stats_reply_handler(self, ev):
        body = ev.msg.body

        if not os.path.isfile(self.flow_stats_file):
            with open(self.flow_stats_file, 'w') as myfile:
                myfile.write("dpid,in_port,out_port,ipv4_src,ipv4_dst,tcp_sr

        with open(self.flow_stats_file, 'a') as myfile:
            dpid = ev.msg.datapath.id
            for stat in body:
                if 'in_port' in stat.match:
                    in_port = stat.match['in_port']
                else:
                    in_port = ''

                try:
                    out_port = stat.instructions[0].actions[0].port
                except:
                    out_port = ''

                if 'ipv4_src' in stat.match:
                    ipv4_src = stat.match['ipv4_src']
                else:
                    ipv4_src = ''

                if 'ipv4_dst' in stat.match:
                    ipv4_dst = stat.match['ipv4_dst']
                else:
                    ipv4_dst = ''

                if 'tcp_src' in stat.match:
                    tcp_src = stat.match['tcp_src']
                else:
                    tcp_src = ''

                if 'tcp_dst' in stat.match:
                    tcp_dst = stat.match['tcp_dst']
                else:
                    tcp_dst = ''

                if 'udp_src' in stat.match:
                    udp_src = stat.match['udp_src']
                else:
                    udp_src = ''
```

```
                if 'udp_dst' in stat.match:
                    udp_dst = stat.match['udp_dst']
                else:
                    udp_dst = ''
                packet_count = stat.packet_count
                byte_count = stat.byte_count
                duration_sec = stat.duration_sec
                myfile.write("%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s\n" % (dpid,i

        self.logger.info('Wrote flow stats to %s', self.flow_stats_file)
```

B.6 configloader.py

```
import collections
import yaml

class ConfigLoader(object):

    @staticmethod
    def load(filename):
        stream = file(filename, 'r')
        dump = yaml.load(stream)
        Config = collections.namedtuple('Config', dump.keys())
        return Config(**dump)
```

B.7 idsbase.py

```
import zope.event

class IdsBase(object):
    '''
    Interactions with specific IDS solutions should
    be implemented in their own classes that
    inherit this class.
    '''

    def ids_alert_subscribe(self, func):
        '''
        Functions that want to receive alerts from the IDS
        should use this method to subscribe.
        '''
        zope.event.subscribers.append(func)

    def ids_alert_fire(self, ev):
        '''
        Classes that inherit this class should use this
        method to notify subscribers that an alert has
        occurred.

        ev should be a Dictionary of fields that contain
```

```
        useful information about the alert such as the
        source and destination addresses.
        '''
        zope.event.notify(ev)
```

B.8 securityonion.py

```
import collections
import re
import socket
from ryu.lib import hub

from morepork.ids import idsbase

class SecurityOnion(idsbase.IdsBase):

    def __init__(self, logger):
        super(SecurityOnion, self).__init__()
        self.logger = logger
        self._alert_regex = re.compile('^<\d{2}>\w{3}\s+\d{1,2}\s\d\d:\d\d:\
        self._alert_class = collections.namedtuple('IdsAlert',
            ['category', 'src_port', 'signature', 'pid', 'server',
            'src_ip', 'dst_port', 'sid', 'date', 'dst_ip',
            'sensor', 'proto'])

    def listen(self, address, port):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.sock.bind((address, port))
        self.ids_thread = hub.spawn(self._server_loop)

    def parse_event(self, data):
        self.logger.info('IDS Alert: '+data)
        m = self._alert_regex.search(data)
        if not m: return None
        alert = self._alert_class(**m.groupdict())
        return alert

    def _server_loop(self):
        self.logger.info('Listening for alerts from IDS')
        while True:
            data, address = self.sock.recvfrom(4096)
            if not data: continue
            alert = self.parse_event(data)
            if not alert: continue
            self.ids_alert_fire(alert)
```

B.9 collector.py

```
class PortStatsCollector(object):
```

```
store = {}

def has_dpid(self, dpid):
    '''
    Check if stats for datapath ID exist
    '''
    return dpid in self.store

def has_port_no(self, dpid, port_no):
    '''
    Check if stats for port exist for a given datapath
    '''
    if not self.has_dpid(dpid):
        return False

    return port_no in self.store[dpid].keys()

def has_metric(self, dpid, port_no, metric):
    '''
    Check if stats for metric exist for given port and datapath
    '''
    if not self.has_port_no(dpid, port_no):
        return False

    if len(self.store[dpid][port_no]) == 0:
        return False

    return hasattr(self.store[dpid][port_no][0], metric)

def ports(self, dpid):
    '''
    Gets a list of ports for a given datapath ID
    '''
    if not self.has_dpid(dpid):
        return None

    return self.store[dpid].keys()

def push(self, dpid, stats):
    '''
    Stores the results from a port stats reply
    '''
    if dpid not in self.store:
        self.store[dpid] = {}

    for set in stats:
        port_no = set.port_no
        if port_no not in self.store[dpid]:
            self.store[dpid][port_no] = []
```

```
            self.store[dpid][port_no].append(set)

            # Limit each list size to 2 items
            if len(self.store[dpid][port_no]) > 2:
                del self.store[dpid][port_no][0]

    def last_value(self, dpid, port_no, metric):
        '''
        Gets the last value for a metric
        '''
        if not self.has_metric(dpid, port_no, metric):
            self.logger.warning('Metric not found: %s port_no=%8x dpid=%016x
                (metric, port_no, dpid))

        vals = self.__get_datapoints(dpid, port_no, metric)
        if vals is not None:
            return vals[1]

    def order1(self, dpid, port_no, metric):
        '''
        Calculates the 1st-order derivative of a metric
        '''
        if not self.has_metric(dpid, port_no, metric):
            self.logger.warning('Metric not found: %s port_no=%8x dpid=%016x
                (metric, port_no, dpid))

        vals = self.__get_datapoints(dpid, port_no, metric)
        if vals is not None:
            return (vals[1] - vals[0]) / vals[2]
        else:
            return 0

    def order2(self, dpid, port_no, metric):
        '''
        Calculates the 2nd-order derivative of a metric
        '''
        if not self.has_metric(dpid, port_no, metric):
            self.logger.warning('Metric not found: %s port_no=%8x dpid=%016x
                (metric, port_no, dpid))

        vals = self.__get_datapoints(dpid, port_no, metric)
        if vals is not None:
            return ((vals[1] - vals[0]) / vals[2]) / vals[2]
        else:
            return 0

    def __get_datapoints(self, dpid, port_no, metric):
        '''
        Gets the last two datapoints and time delta
```

```
        '''
        if (dpid not in self.store or
            port_no not in self.store[dpid].keys() ):
            return None

        datasets = self.store[dpid][port_no][-2:]
        if len(datasets) < 2:
            return None

        if (not hasattr(datasets[0], metric) or
            not hasattr(datasets[1], metric)):
            return None

        a = getattr(datasets[0], metric)
        b = getattr(datasets[1], metric)
        dt = datasets[1].duration_sec - datasets[0].duration_sec
        return [a, b, dt]
```

B.10 tripwire.py

```
import os

from collections import namedtuple
from ryu.controller import event

from morepork.config import configloader

class Tripwire(object):

    ENV_TRIPWIRE_CONF = 'TRIPWIRE_CONF'

    def __init__(self, logger, collector):
        self.logger = logger
        self.collector = collector

        self._load_environment()
        self.CONF = configloader.ConfigLoader.load(self._tripwire_conf)

        self._wire = namedtuple('Wire', 'dpid, port_no, metric, value, thresho

    def _load_environment(self):
        if self.ENV_TRIPWIRE_CONF in os.environ:
            self._tripwire_conf = os.environ[self.ENV_TRIPWIRE_CONF]
        else:
            self.logger.error('Environment variable not found: %s' %
                self.ENV_TRIPWIRE_CONF)
            self._tripwire_conf = None

    def process_flow_stats(self):
        '''
```

```
        Run threshold checks on flow stats
        '''
        return [] # TODO

    def process_port_stats(self):
        '''
        Run threshold checks on port stats
        '''
        conf = self.CONF.port
        col = self.collector

        results = []

        for dpid, ports in conf.iteritems():
            if not col.has_dpid(dpid): continue

            for port_no, metrics in ports.iteritems():
                if not col.has_port_no(dpid, port_no): continue

                for metric, settings in metrics.iteritems():
                    if not col.has_metric(dpid, port_no, metric): continue

                    if 'threshold' not in settings:
                        self.logger.warning('Threshold not defined for port-
                            (metric, port_no, dpid))
                        continue

                    if 'derivative' in settings:
                        derivative = settings['derivative']
                    else:
                        derivative = 0

                    if settings['derivative'] == 2:
                        val = col.order2(dpid, port_no, metric)
                    elif settings['derivative'] == 1:
                        val = col.order1(dpid, port_no, metric)
                    else:
                        val = col.last_value(dpid, port_no, metric)

                    threshold = settings['threshold']
                    tripped = val >= threshold

                    results.append(self._wire(dpid, port_no, metric, val, th

        return results
```

Appendix C

Example flow tables

This is an example of the flow tables ina switch when suspicious traffic is detected and port mirroring flow entries have been added to the tables.

```
stats_reply (xid=0xe66ef4ff): flags=none type=1(flow)
cookie=0, duration_sec=114s, duration_nsec=772000000s, table_id=0, priority=0, n
cookie=0, duration_sec=84s, duration_nsec=305000000s, table_id=1, priority=1, n_
cookie=0, duration_sec=114s, duration_nsec=772000000s, table_id=1, priority=0, n
cookie=0, duration_sec=84s, duration_nsec=721000000s, table_id=2, priority=1, n_
cookie=0, duration_sec=84s, duration_nsec=720000000s, table_id=2, priority=1, n_
cookie=0, duration_sec=114s, duration_nsec=773000000s, table_id=2, priority=0, n
```

Appendix D

IDS configuration and code

This is the configuration for *syslog-ng* — the program that reads the alert log from Sguil and `ids.py` and forwards the alerts to Morepork. The code for `ids.py` follows the configuration.

D.1 syslog-ng.conf in Security Onion

```
# On your master server (running sguild), configure /etc/syslog-ng/syslog-ng
# with a new "source" to monitor /var/log/nsm/securityonion/sguild.log for "
# Received" lines and a new "destination" to send to your external system, a
# then restart syslog-ng. To do this modify /etc/syslog-ng/syslog-ng.conf an
# the following lines:

# This line specifies where the sguild.log file is located, and informs syslo
# to tail the file, the program_override inserts the string sguil_alert into
# string

source s_sguil { file("/var/log/nsm/securityonion/sguild.log" program_overri

# This line filters on the string  Alert Received

filter f_sguil { match("Alert Received"); };

# This line tells syslog-ng to send the data read to the IP address of the M
# instance, via UDP to port 514

destination d_sguil_udp { udp("192.168.0.150" port(514)); };

# This log section tells syslog-ng how to structure the previous  source /
# destination  and is what actually puts them into play

log {
    source(s_sguil);
    filter(f_sguil);
    destination(d_sguil_udp);
};
```

D.2 ids.py

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

from datetime import datetime
import subprocess as sub
import re
import sys

try:
    file = None
    p = None
    while True:
        file = open('/var/log/nsm/securityonion/sguild.log', 'a')
        p = sub.Popen(('sudo', 'tcpdump', '-nn', '-l', '-ieth0'), stdout=sub.
        for row in iter(p.stdout.readline, b''):
            m = re.search('(?P<src_ip>\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\.(?
            src = m.group('src_ip')
            dst = m.group('dst_ip')
            sport = m.group('sport')
            dport = m.group('dport')

            t = datetime.today()
            t1 = t.strftime('%b %d %H:%M:%S')
            t2 = t.strftime('%Y-%m-%d %H:%M:%S')

            out = '<13>%s 489-securityonion2 sguil_alert: %s pid(3022)
Alert Received: 0 3 misc-activity 489-securityonion2-eth0 {%s} 6 38 {} %s %s %s %s
                t1, t1, t2, src, dst, 17, sport, dport)
            print out
            file.write(out)
except KeyboardInterrupt:
    print "Quitting..."
finally:
    if p:
        p.terminate()            # zombie protection, if needed
    if file:
        file.close()
    sys.exit()
```

Appendix E

Mininet automated test scripts

This appendix shows the Mininet topology file used during the tests. The iteration script follows after the topology file with an example output of the iteration script during a test run. Note that the longer lines are truncated.

E.1 mininet-topo.py

```
"""
Written by Ewen McNeill <ewen2naos.co.nz>, 2014-07-17

Adapted by Ben Vidulich <ben@vidulich.co.nz>, 2014-10-28
"""

import datetime
import os
import re
import sys
import subprocess
import time
import threading

from mininet.net  import Mininet
from mininet.node import OVSSwitch, RemoteController
from mininet.topo import Topo
from mininet.link import Intf
from mininet.log  import setLogLevel
from mininet.cli  import CLI
from mininet.util import run

setLogLevel('info')
#setLogLevel('debug')    # For diagnostics

def checkIntf( intf ):
    "Make sure intf exists and is not configured."
    if ( '%s:' % intf ) not in quietRun( 'ip link show' ):
        error( 'Error:', intf, 'does not exist!\n' )
        exit( 1 )
```

```
    ips = re.findall( r'\d+\.\d+\.\d+\.\d+', quietRun( 'ifconfig ' + intf ) )
    if ips:
        error( 'Error:', intf, 'has an IP address,'
               'and is probably in use!\n' )
        exit( 1 )

class MeasurementThread(threading.Thread):

    port_stats_file = 'port-stats-%s.csv'

    def __init__(self, switch):
        super(MeasurementThread, self).__init__()
        self.switch = switch
        self._stop = threading.Event()

        if 'TAG' in os.environ:
            self.tag = os.environ['TAG']
        else:
            self.tag = ''

        self.dump_pattern = re.compile(r'port\s+(?P<port>.*): rx pkts=(?P<rx_p

        t = datetime.datetime.today()
        self.start_time = int(t.strftime('%s'))
        self.port_stats_file = self.port_stats_file % self.start_time

        if not os.path.isfile(self.port_stats_file):
            with open(self.port_stats_file, 'w') as myfile:
                myfile.write("duration_sec,tag,port,rx_bytes,rx_pkts,tx_bytes,

    def run(self):
        while not self.stopped():
            time.sleep(1)
            t = datetime.datetime.today()
            now = int(t.strftime('%s'))
            duration = now - self.start_time
            dump = self.switch.dpctl('dump-ports')
            matches = [m.groupdict() for m in re.finditer(self.dump_pattern, d
            with open(self.port_stats_file, 'a') as myfile:
                for m in matches:
                    myfile.write("%s,%s,%s,%s,%s,%s,%s\n" % (duration, self.ta

    def stop(self):
        self._stop.set()

    def stopped(self):
        return self._stop.isSet()

class MoreporkOneSwitch(Topo):
    "Morepork One Switch example topology"
```

```
    def __init__(self):
        super(MoreporkOneSwitch, self).__init__()

        # Add hosts and switches
        self.srcHost  = self.addHost('h1', ip='10.0.0.1' )
        self.dstHost = self.addHost('h2', ip='10.0.0.2' )
        oneSwitch = self.addSwitch('s1', dpid='0000000000000099',
                                   listenPort=6634,
                                   protocols='OpenFlow13' )

        # Add links
        self.addLink( self.srcHost, oneSwitch )
        self.addLink( oneSwitch, self.dstHost )

# try to get hw intf from the command line; by default, use eth0
intfName = sys.argv[ 1 ] if len( sys.argv ) > 1 else 'eth0'

sw = MoreporkOneSwitch()
ryu = RemoteController('ryu', ip='127.0.0.1', port=6633 )
net = Mininet(topo=sw, switch=OVSSwitch, build=False)
net.addController(ryu)
net.build()

# Add physical interface
switch = net.switches[ 0 ]
_intf = Intf( 'eth0', node=switch )
_intf2 = Intf( 'eth2', node=switch )

net.start()
run("ovs-vsctl set bridge s1 protocols=OpenFlow10,OpenFlow13")

thread1 = MeasurementThread(switch)
thread1.daemon = True
thread1.start()

print 'Warming up...'
time.sleep(30)
print 'Beginning attack...'
dstHost, srcHost = net.getNodeByName('h1', 'h2')
dstHost.cmd('iperf -s -p 53 -u &')
srcHost.cmd('iperf -c ', dstHost.IP(), ' -p 53 -u -t 120') # Run for 2 minut
print 'Attack complete...'
time.sleep(30)
print 'Cool-down complete...'

thread1.stop()
net.stop()
thread1.join(1.0)
```

E.2 mn-run.sh

```
#!/bin/bash

export count=1
for run in {1..50}
do
    sudo mn -c > /dev/null 2>&1
    echo "----------- BEGIN TEST $count ----------"
    sudo TAG=$count python ~/engr489-project/tools/mininet-topo.py
    echo "-----------  END TEST $count  ----------"
    count=$((count+1))
done
```

E.3 Example output

```
$ ./mn-run.sh
-----=====  BEGIN TEST 1 =====-----
*** Creating network
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1
Warming up...
Beginning attack...
Attack complete...
Cool-down complete...
*** Stopping 1 controllers
ryu
*** Stopping 1 switches
s1 ..
*** Stopping 2 links

*** Stopping 2 hosts
h1 h2
*** Done
-----=====  END TEST 1  =====-----
```

Appendix F

Spark's comment on their September 2014 issues

The following statement was provided by Spark for this project on October 28, 2014 with regards to their DNS issues from September 2014.

In our case New Zealand ISPs and customers were essentially used as pawns in an attack between several overseas parties. We know that it caused major inconveniences for many of our customers for a few days, and weve been up front and apologised for what occurred. We also pointed out that the threat of cyber-criminal activity is very real. Its an industry wide and global issue. Every year Spark New Zealand, like all other ISPs, and indeed any organisation with a large computer network, must deal with many attacks, large and small.

In a typical year we deal with thousands of potential attacks the overwhelming majority of these are intercepted before they can impact customers. In a globally connected world, the risk of such attacks has increased, and they are becoming more and more sophisticated and well-resourced. And the nature of the attacks continues to evolve dynamically. In this particular case, the bad guys found a new way through.

Spark spends around $400 million every year keeping the networks going. We have a state of the art network operating centre that we use to monitor for and control network operations. This is where crises such as the DNS outage are managed from.

For obvious reasons we cant disclose the details of the security measures we took. However we are constantly reviewing and monitoring our systems and reviewing what is happening internationally to keep ahead. Today, there is over 10 billion connected devices around the world, projected to be over 25 billion by the end of the decade, so the avenues of attack for those who want use the internet maliciously are much wider and more pervasive.

That said, New Zealand is a relatively safe digital nation. By international standards we have robust and reliable internet infrastructure and the incidence of cyber-attacks is lower than in many other nations. While ISPs will always detect and repel the vast majority of attacks, this attack is a useful reminder that good end-user security of devices and connections is also an important way to combat these attacks.

We constantly remind people to keep their internet device security up to date, conduct regular scans and regularly update the operating software and firmware on home networks. And dont click on suspicious links or download files when you are not sure of the contents.

F.1 The DNS Issue

F.1.1 What happened?

Cyber criminals based overseas appear to have been attacking web addresses in Eastern Europe, and were bouncing the traffic off Spark customer connections, in what is known as a distributed denial of service (DDoS) attack.

The DDoS attack was dynamic, predominantly taking the shape of an amplified DNS attack which means an extremely high number of connection requests in the order of thousands per second - were being sent to a number of overseas web addresses with the intention of overwhelming and crashing them. Each of these requests, as it passes through our network, queries our DNS server before it passes on so our servers were bearing the full brunt of the attack.

While the Spark network never crashed, we did experience extremely high traffic loads hitting our DNS servers which meant many customers had either slow or at times no connectivity (as their requests were timing out).

There were multiple attacks, which were dynamic in nature. They began on Friday night, subsided, and then began again early Saturday, continuing over the day. By early Sunday morning traffic levels were back to normal. We saw the nature of the attack evolve over the period, possibly due to the cyber criminals monitoring our response and modifying their attack to circumvent our mitigation measures in a classic whack a mole scenario.

F.1.2 How did they get access through the Spark Network?

During the attack, we observed that a small number of customer connections were involved in generating the vast majority of the traffic. This was consistent with customers having malware on their devices and the timing coincided with other DNS activity related to malware in other parts of the world.

However, while we believe malware was as a potential factor, we also identified that cyber criminals had been accessing vulnerable customer modems on our network. These modems have been identified as having open DNS resolver functionality, which means they can be used to carry out internet requests for anyone on the internet. This makes it easier for cyber criminals to bounce an internet request off them (making it appear that the NZ modem was making the request, whereas it actually originates from an overseas source).

Most of these modems were not supplied by Spark and tend to be older or lower-end modems. What remains clear is that good end user security remains an important way to combat these attacks. With the proliferation of devices in households, that means both the security within your device and the security of your modem.

F.1.3 What did Spark do?

We disconnected those modems from our network and contacted all the affected customers. We have also taken steps at a network level to mitigate this modem vulnerability. We also scanned our entire broadband customer base to identify any other customers who may be using modems with similar vulnerabilities and contacted those customers to advise them on what they should do.

With respect to malware we continue to strongly encourage our customers to keep their internet device security up to date, conduct regular scans and regularly update the operating software and firmware on their home network. We also continue to advise customers not to click on suspicious links or download files when they are not sure of the contents.

We have also taken steps at the network level to make it more difficult for cyber criminals to exploit the DNS open resolver modem vulnerability and were using the latest technology to strengthen our network monitoring and management capabilities. For security reasons we cant detail these steps, however this is an ongoing battle to stay one step ahead of cyber criminals who are continually using more and more sophisticated tactics.

One of the mitigation options in response to the DDoS attacks involves blocking port 53, which effectively stops one of the means for some customer devices and modems to be misused. Were aware other ISPs did the same thing to combat this latest development in cyber-threats. However in certain cases blocking port 53 does have other impacts on connectivity. So since the weekend weve been continuing to make enhancements and changes.

As part of these enhancements, we took some further steps to enable us to better look under the hood across some parts of the network. While the initial measures taken had largely mitigated the impact of the attacks, we didnt have total visibility of everything that was going on, especially in terms of abnormal traffic patterns.

Within the first hour of taking these further steps we saw DNS traffic coming from just three of our home broadband customers representing 4% of our total DNS traffic for that period. One connection alone had 1.2 million DNS requests in an hour. As we have port 53 blocked, we believe that this may be due to malware previously installed on these customers devices. We dont believe this is a new attack, its likely the malware was installed before the attack.

During the weekend issues that among other things we saw incoming traffic being bounced off a number of vulnerable customer modems (those with DNS open resolver functionality). Further insights did not involve any significant level of incoming traffic, which tends to point to device malware, rather than a specific modem issue. This demonstrates there were a number of different vectors involved in the weekends DDoS attacks.

This is just one vivid illustration of the potential scale of cyber-threats and the impact that can be generated from just a very small number of connections. Like all ISPs we see evidence of literally thousands of attacks every year and the vast majority of these never impact on the customer experience across our network because of proactive management.

F.1.4 Why only Spark?

We cant say what other networks experienced. However, cyber criminals often look for clusters of IP addresses to use in any particular DDoS attack. That makes it more likely that these IP addresses belong to the customers of a single ISP even more likely with a large ISP like Spark.

Bibliography

[1] ANDERSON, R. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2008.

[2] BAMBENEK, J. A Chargen-based DDoS? Chargen is still a thing? `https://isc.sans.edu/diary/A+Chargen-based+DDoS%3F+Chargen+is+still+a+thing%3F/15647`. [Online; accessed 20-Oct-2014].

[3] BURKS, D. Security Onion. `http://blog.securityonion.net/p/securityonion.html`. [Online; accessed 31-Oct-2014].

[4] CANTONI, M. Honeypot DNS and amplification attacks. `http://www.nothink.org/honeypot_dns.php`. [Online; accessed 21-Oct-2014].

[5] CORNELL, D. DNS Amplification Attacks. `http://labs.opendns.com/2014/03/17/dns-amplification-attacks/`. [Online; accessed 21-Oct-2014].

[6] DUBENDORFER, T., BOSSARDT, M., AND PLATTNER, B. Adaptive distributed traffic control service for ddos attack mitigation. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International* (April 2005), pp. 8 pp.–.

[7] EASTEP, T. M. Simple way to set up Split DNS. `http://shorewall.net/SplitDNS.html`. [Online; accessed 21-Oct-2014].

[8] EVANS, C. C. YAML. `http://www.yaml.org/`. [Online; accessed 30-Oct-2014].

[9] FERGUSON, P., AND SENIE, D. BCP 38 - Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, May 2000.

[10] GRAHAM-CUMMING, J. Understanding and mitigating NTP-based DDoS attacks. `http://blog.cloudflare.com/understanding-and-mitigating-ntp-based-ddos-attacks/`. [Online; accessed 22-Oct-2014].

[11] KAMBOURAKIS, G., MOSCHOS, T., GENEIATAKIS, D., AND GRITZALIS, S. A fair solution to dns amplification attacks. In *Digital Forensics and Incident Analysis, 2007. WDFIA 2007. Second International Workshop on* (Aug 2007), pp. 38–47.

[12] KEALL, C. RAW DATA: Spark releases Q+A on outages, cyber-attack. `http://www.nbr.co.nz/article/raw-data-spark-releases-qa-outages-cyber-attack-ck-162069`. [Online; accessed 22-Oct-2014].

[13] KONG, J., MIRZA, M., SHU, J., YOEDHANA, C., GERLA, M., AND LU, S. Random flow network modeling and simulations for ddos attack mitigation. In *IEEE International Conference on Communications, 2003. ICC '03* (May 2003), vol. 1, pp. 487–491 vol.1.

[14] MAHAJAN, R., BELLOVIN, S. M., FLOYD, S., IOANNIDIS, J., PAXSON, V., AND SHENKER, S. Controlling high bandwidth aggregates in the network. *SIGCOMM Comput. Commun. Rev. 32*, 3 (July 2002), 62–73.

[15] MICROSOFT. What is a botnet? `http://www.microsoft.com/security/resources/botnet-whatis.aspx`. [Online; accessed 20-Oct-2014].

[16] MIMOSO, M. Spamhaus DDoS Attacks Triple Size of Attacks on US Banks. `http://threatpost.com/spamhaus-ddos-attacks-triple-size-attacks-us-banks-032713/77675`. [Online; accessed 22-Oct-2014].

[17] MININET TEAM. Mininet. `http://mininet.org/`. [Online; accessed 31-Oct-2014].

[18] MORROW, C. Customer Blackhole Community. `http://www.he.net/adm/blackhole.html`. [Online; accessed 21-Oct-2014].

[19] NETWORKS, A. Digital Attack Map. `http://www.digitalattackmap.com/`. [Online; accessed 27-Oct-2014].

[20] NIST. Vulnerability Summary for CVE-2013-5211. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-5211`. [Online; accessed 22-Oct-2014].

[21] NLANR/DAST. Iperf. `https://iperf.fr/`. [Online; accessed 31-Oct-2014].

[22] OF HOMELAND SECURITY, D. DNS Amplification Attacks. `https://www.us-cert.gov/ncas/tips/ST04-015`. [Online; accessed 20-Oct-2014].

[23] OF HOMELAND SECURITY, D. DNS Amplification Attacks. `https://www.us-cert.gov/ncas/alerts/TA13-088A`. [Online; accessed 19-Oct-2014].

[24] OF HOMELAND SECURITY, D. UDP-based Amplification Attacks. `https://www.us-cert.gov/ncas/alerts/TA14-017A`. [Online; accessed 20-Oct-2014].

[25] OPEN NETWORKING FOUNDATION. Software-Defined Networking (SDN) Definition. `https://www.opennetworking.org/sdn-resources/sdn-definition`. [Online; accessed 21-Oct-2014].

[26] OPEN VSWITCH. Open vSwitch. `http://openvswitch.org/`. [Online; accessed 31-Oct-2014].

[27] ORACLE. VirtualBox. `https://www.virtualbox.org/`. [Online; accessed 31-Oct-2014].

[28] PBUG. Tfreak. `http://hackepedia.org/?title=Tfreak`. [Online; accessed 20-Oct-2014].

[29] PFAFF, B., LANTZ, B., HELLER, B., BARKER, C., BECKMANN, C., COHN, D., TALAYCO, D., ERICKSON, D., MCDYSAN, D., WARD, D., CRABBE, E., GIBB, G., APPENZELLER, G., TOURRILHES, J., TONSING, J., PETTIT, J., YAP, K., POUTIEVSKI, L., VICISANO, L., CASADO, M., TAKAHASHI, M., KOBAYASHI, M., YADAV, N., MCKEOWN, N., DHEUREUSE, N., BALLAND, P., RAMANATHAN, R., PRICE, R., SHERWOOD, R., DAS, S., GANDHAM, S., YABE, T., YIAKOUMIS, Y., AND KIS, Z. L. OpenFlow Switch Specification. Specification, Open Networking Foundation, June 2012.

[30] PORTER, R. DDoS and BCP 38. `https://isc.sans.edu/diary/DDoS+and+BCP+38/17735`. [Online; accessed 21-Oct-2014].

[31] PRINCE, M. Deep Inside a DNS Amplification DDoS Attack. http://blog.cloudflare.com/deep-inside-a-dns-amplification-ddos-attack/. [Online; accessed 20-Oct-2014].

[32] PRINCE, M. Technical Details Behind a 400Gbps NTP Amplification DDoS Attack. http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack/. [Online; accessed 22-Oct-2014].

[33] PRINCE, M. The DDoS That Knocked Spamhaus Offline (And How We Mitigated It). http://blog.cloudflare.com/the-ddos-that-knocked-spamhaus-offline-and-ho/. [Online; accessed 22-Oct-2014].

[34] PROLEXIC. DoS and DDoS Glossary of Terms. http://www.prolexic.com/knowledge-center-dos-and-ddos-glossary.html. [Online; accessed 20-Oct-2014].

[35] PROLEXIC. DDoS Boot Camp: Basic Training for an Increasing Cyber Threat.

[36] REKHTER, Y., MOSKOWITZ, B., KARRENBERG, D., DE GROOT, G., AND LEAR, E. Address Allocation for Private Internets. RFC 1918, February 1996.

[37] RYAN, P., AND SCHNEIDER, S. A. *The modelling and analysis of security protocols: the csp approach*. Addison-Wesley Professional, 2001.

[38] RYU SDN FRAMEWORK COMMUNITY. Ryu. http://osrg.github.io/ryu/. [Online; accessed 31-Oct-2014].

[39] SUN, C., LIU, B., AND SHI, L. Efficient and low-cost hardware defense against dns amplification attacks. In *IEEE Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008.* (Nov 2008), pp. 1–5.

[40] TABLEAU SOFTWARE. Tableau. http://www.tableausoftware.com/. [Online; accessed 31-Oct-2014].

[41] TECHNET, M. Disable Recursion on the DNS Server. http://technet.microsoft.com/en-us/library/ee649165(v=ws.10).aspx. [Online; accessed 21-Oct-2014].

[42] VIXIE, P. DNS Response Rate Limiting (DNS RRL). TN 1, April 2012.

[43] VIXIE, P. Rate-limiting State: The edge of the Internet is an unruly place. *ACMQueue 12*, 2 (February 2014).

[44] WINSTEAD, E. Response Rate Limiting with BIND. APRICOT 2014, 2014.