Traffic data processing using large scale graph processing systems

Yorin Anne De Jong

Master of Telematics - Communication Networks and Networked Services (2
Submission date: July 2014
Supervisor: Yuming Jiang, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Traffic data processing usinglarge scale graph processing
Student: Yorin Anne de Jong

Problem description:

Computers connected to the internet are subject to being infected with different kinds of malware. Often, an infected computer will try to infect more computers, launch attacks, send spam or produce other malicious traffic. The general scope of the thesis work is to investigate method(s) to automatically detect such malicious traffic from large datasets (e.g. in the order of hundreds of gigabytes of metadata) by means of anomaly detection. The specific goal of the thesis work is to investigate different methods to store NetFlow data in a scalable graph structure, and subsequently try different graph processing systems on the NetFlow data to detect malicious traffic. It is expected that the work will use at least one of the available graph processing frameworks to detect malicious traffic on the network. NetFlow data is provided by UNINETT and consists of metadata of traffic but not payload (e.g. the fact that two IP addresses exchanged packets, but not their contents).

Responsible professor: Yuming Jiang, ITEM
Supervisor: Gurvinder Singh, UNINETT

Abstract

Anomaly detection in internet traffic today is largely based on quantifying traffic data. This thesis proposes a new algorithm SpreadRank, which detects spreading of internet traffic as an additional metric for traffic anomaly detection. SpreadRank uses large scale graph processing to calculate spreading from multiple gigabytes of NetFlow data obtained from core routers. Studying spreading is a useful tool in determining the role of an end-host and in identifying malicious behaviour.

Sammendrag

Anomalideteksjon i nettrafikk i dag er stortsett basert på mengder trafikk. Denne avhandlingen introduserer en ny algoritme SpreadRank, som vil detektere spredning i nettrafikk som et ekstra målepunkt for anomalideteksjon. SpreadRank vil bruke flere gigabytes i NetFlow logg fra core-rutere i storskala grafprosessering for å beregne spredning. Spredning er et nyttig verktøy for å undersøke hva slags rolle en datamaskin på nettet har og for å finne skadelig oppførsel.

Preface

Writing this Master's thesis has been an educative experience. I would like to gratefully acknowledge the guidance, support and encouragement of my supervisors, Yuming Yiang from the Norwegian University of Science and Technology (NTNU) and Gurvinder Singh from UNINETT AS, as well as Tor Gjerde from UNINETT for his continued availability, enthusiasm and willingness to provide data and expert opinions. My gratitude extends to Arne Øslebø and Rune Sydskjør at UNINETT for providing me with the opportunity to check my findings against UNINETT's logging systems.

Finally, I would like to thank my family for giving me support throughout, and to my friends in Norway for being there when I needed them, and for inviting me to dinner when I was too busy to cook.

Contents

List of Figures

List of Algorithms

List of Terms

client	An entity on the network that initiates connections.
contact port	Fixed port used to contact a service.
denial of service	A method for making a service unavailable to its users.
depth	The longest path between hosts that copy each others behaviour.
ephemeral port	Short-lived automatically allocated temporary port.
Giraph	Open source graph processing system.
graph	Mathematical structure consisting of vertices and edges.
host	An entity on the network, indicated by and IP address.
natural spreading	Spreading is the intended behaviour of the protocol.
NetFlow	Proprietary data format from Cisco describing traffic flows.
network layer	3rd layer in the OSI model, the internet is a network where hosts are interconnected via routers.
nfdump	Open source utility to convert NetFlow to text.

server	An entity on the network that answers connection requests.
service	An IP address and port number pair.
spreading	Score for a host indicating how far traffic sent from said host spreads.
transport layer	4th layer in the OSI model, the internet is a network where hosts can communicate directly with each other.
well-known port	Fixed port that has been allocated by the IANA for a specific service.

List of Acronyms

Bash Bourne Again Shell.

BGP Border Gateway Protocol.

CSV Comma Separated Value.

DNS Domain Name System.

DoS Denial of Service.

HDFS the Hadoop File System.

HTTP HyperText Transfer Protocol.

HTTPS HyperText Transfer Protocol Secure.

NTNU the Norwegian University of Science and Technology.

NTP Network Time Protocol.

SMTP Simple Mail Transfer Protocol.

SSH Secure Shell.

Chapter 1

Introduction

The internet today is a very important communication network. Nearly everybody is connected to it, many are dependent on it, and the possibilities are virtually endless. The applications range from looking at pictures of cats, to high intensity stock trading and remote surgeries; however, not all applications are well intended towards other internet users. Criminals write malware that will send spam and spy on people's computers to look for credit card information. Activists use botnets to bombard sites they don't like, in order to make them unavailable.

Internet providers want to block clients that participate in harmful activity, but this is not a simple task. The amount of traffic that passes through a back bone router is huge, and special provisions need to be in place to log this. Cisco is a large supplier of router hardware, and has introduced the NetFlow format [5]. NetFlow data does need to be stored an processed, for which multiple solutions are available [25, 15].

Current software for analysing NetFlow logs is based on counting flows. A relatively high number of flows in a short period of time is reason for alarm, and a high number of flows for a single IP address may be a reason to investigate. The challenge is that a high number of flows does not necessarily indicate a problem.

1.1 Anomaly detection

An anomaly is a non-conforming pattern [3]. There are multiple methods to detect anomalous traffic. It is possible to detect high or low resource consumption [11], standards-conforming or non-standards-conforming packets [8], or asymmetric traffic which should be symmetric [10]. However, these anomalies do not necessary indicate malicious activity. Non-standard packets may be sent between hosts running legacy and/or proprietary software, asymmetric traffic may be caused by network scanning tools, improperly configured firewalls or IP spoofing, and high-rate traffic can just as well indicate a popular server, as an attack [7].

This thesis will focus on spreading of traffic as an anomaly, as malicious traffic is intended to spread as far as possible, while bona fide traffic will often remain between two hosts. To illustrate: a criminal who created malware that steals credit cards wants it installed on as many computers as possible. The owner of a botnet wants as many computers as possible to join it. An end user sending an e-mail or doing online banking, however, will want this information to be only available to the intended recipients.

1.2 Graph processing

Graph processing is often associated with data mining, due to Facebook actively working on the technology to mine social data [4], though it has a lot of different uses. Some examples of graph processing in real-world applications are navigation (roads as edges), financial (currency flow), internet search (Google pagerank [17]) and rumour spreading [16]. This thesis will use graphs to process network data from NetFlow, which is a graph that shows information flow.

The NetFlow data will be provided by UNINETT, which administers the core routers in the Norwegian academic network. Backbone NetFlow traffic is typically hundreds of megabytes to gigabytes of flow data per day. By observing such a large network, it is possible to look for patterns. Observing individual network nodes may give some indication of the type of traffic being sent, though observing the interaction between many different hosts can show information that is not visible by only observing a few hosts.

1.3 Privacy

This thesis is based on sampled NetFlow data from December 2013, made available by UNINETT. NetFlow data contains, among others, source IP, destination IP, protocol, and time, but no payload information. Sampled NetFlow is a NetFlow log where not everything is logged, UNINETT logs about 1 % of flows, this is done for performance and storage reasons. Even though no payload information was made available for this study, payload information can be helpful when identifying traffic. There are known techniques for using payload information without infringing on privacy [18].

Chapter 2 NetFlow

2.1 Format

The traffic data provided by UNINETT is stored in the NetFlow File Format [5]. This format consists of flows, which are data units consisting of information from the network layer and transport layer, for example IP addresses, transport protocol, and port numbers where applicable. NetFlow can be converted to human readable text using the `nfdump` program[1].

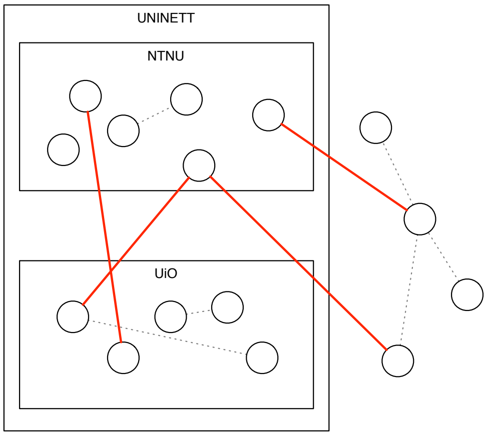
2.2 Available information

NetFlow data is logged on the core routers, which means that only traffic that passes through these routers is observable. UNINETT peers with most Norwegian education institutions, which have their own routers connected to the core routers. This means that UNINETT can only observe traffic that happens between institutions, or between institutions and hosts on the outside. In other words, traffic between hosts within an institution is not observable for UNINETT.

Figure 2.1 shows some hosts in different networks, and some flows between them. The figure illustrates how only flows that go through UNINETT's core routers will show up in the NetFlow log. Flows between hosts within an institution will never pass through UNINETT's core routers, even though the institution is connected to the internet through UNINETT. Flows completely outside of UNINETT's core network are not logged either, naturally. Flows that are between one host at an institution connected through UNINETT – for example NTNU – and one host in another institution, can be logged by UNINETT's core routers, because the traffic must pass through these routers.

[1] http://nfdump.sourceforge.net

Figure 2.1: Illustration of which flows are logged



2.3 Direction

Flows are between a source and destination IP address. One TCP or UDP session will usually be observed as multiple flows, because NetFlow logs are written by core routers operating on the network layer, which will not keep track of TCP of UDP connections, as this is a transport layer function. In order to observe spreading, it is important to know which host initiated the connection. A possible solution would be to take the first flow between two hosts, and merge it with all subsequent flows that are between the same IP addresses and port numbers, until a time-out occurs. However, for performance reasons, UNINETT makes use of sampled NetFlow data, which means that only 1 % of all the packets are logged. This makes it harder to reliably establish which party initiated a connection, as the first observed flow may be in the opposite direction from the actual first flow.

TCP and UDP are transport protocols which offer end-to-end communication between hosts. In order to allow hosts to run multiple services, these protocols use a two byte *port number* to indicate the type of service. In order to set up a TCP or UDP connection, a client must first select a port number at random. This port is called the *ephemeral port*, and its range differs per operating system, but it is never lower than 1024. The client will then send an initial packet to the server, from the *ephemeral port*, to the *contact port*. The *contact port* will often be a *well-known port*, which is a port number under 1024. The direction of a flow can therefore be assumed to be initiated by the party with the lowest port number. The list of well-known ports is maintained by IANA [20].

Chapter 3
Graph processing

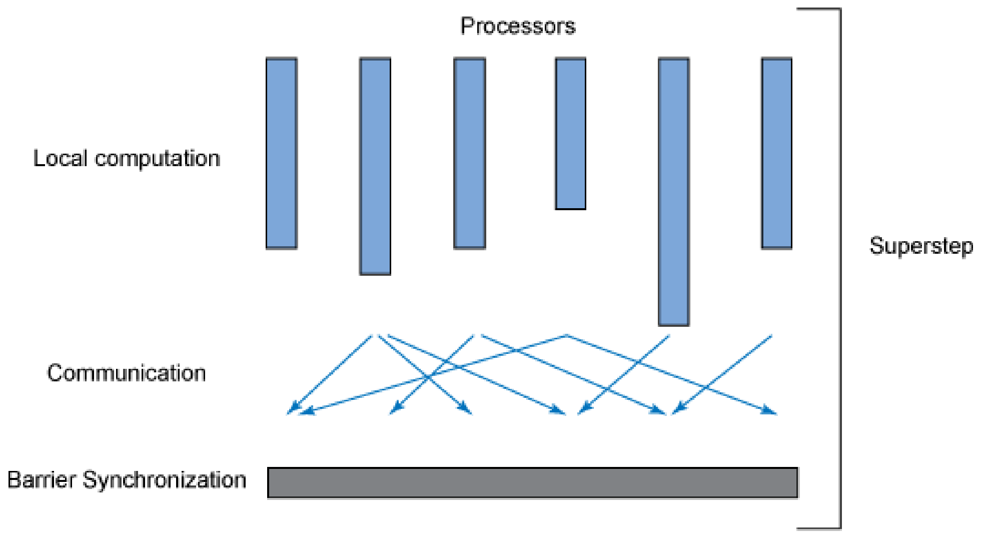
In order to process large graphs, a graph processing system is required. A graph processing system uses a data model of vertices which are connected by edges [21]. Vertices can have an ID and a value, edges can have a direction and a value. The vertex values are modified during computation, and edges can be added and/or removed during computation. A graph system can scale out horizontally (linearly with size), and is therefore able to process graph data at large scale. A large graph system must also be able to handle failover, such as network, storage or hardware failure. There are multiple graph systems available, for example GraphLab [13], GraphX [24], Pregel [14] and Giraph [1, 4]. This thesis will focus on Giraph, but other systems are expected to give similar results.

3.1 Bulk Synchronous Parallel model

By running calculations in parallel, more computations per second are possible. The case for parallel computing is strengthened by the fact that processing power is cheap nowadays, but fast processing is achieved by using many processors in parallel. Parallel computing does, however, pose an important challenge, in that a single processor cannot share memory with another processor, and synchronisation is expensive, as it requires processors to wait, as memory cannot be reliably changed by two different processes.

The Bulk Synchronous Parallel model [22] (figure 3.1) is a model for parallel computing. It consists of components that handle processing and/or memory. Operations are divided in so-called supersteps. A router allows components to exchange point to point messages at the end of a superstep execution, which allows for storage access between components. The model is useful for running large calculations that will take a very long time with standard hardware.

Figure 3.1: BSP programming model [21]



3.2 Giraph

Giraph is a software based implementation of the the Bulk Synchronous Parallel model, and allows for a large amount of physical machines to form a cluster. A Giraph computation program consists of a single `calculate` function, which has two arguments; (1) the vertex and (2) the messages that were sent to the vertex during the last superstep. During every superstep of the computation, a vertex may send a message to another vertex via one of its outgoing edges. A vertex may also “vote to halt”, to indicate that it is done calculating. If a vertex votes to halt, it will not be called during subsequent supersteps, unless it receives a message from another vertex. A vertex that has received a message will always be called, and any previous votes to halt are forgotten; the vertex will have to vote to halt again.

Before the computation is started, the graph is read by a `VertexInputFormat` or an `EdgeInputFormat`. After computation, the graph is written by a `VertexOutputFormat` or `EdgeOutputFormat`[1]. When the program is started, Giraph will start worker processes on the physical machines. The workers will each read a part of the graph, Giraph will distribute the graph, workers will execute supersteps and after the last step is completed, the workers write their part of the graph.

Giraph takes care of the communication between the workers, and distributes the workload over the workers. Due to the nature of BSP, it is important for Giraph that every worker will take roughly the same amount of time to complete a superstep.

[1]https://giraph.apache.org/io.html

After each superstep, the nodes have to synchronise and start working on the next superstep, which means that if one worker is slow, all other workers have to wait for it to finish.

3.3 Design choices

3.3.1 Optimisations

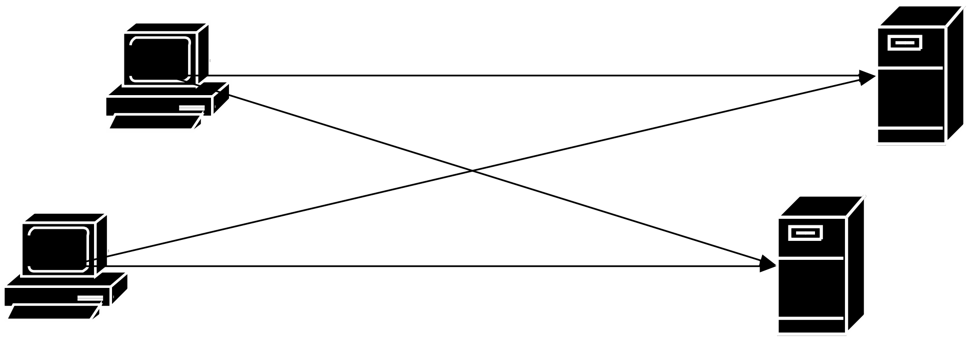
When implementing a computation for a graph system, such as Giraph, it is important to make the run-time of each computation as short as possible. During each superstep, all workers will run computations, so that each vertex is computed once per superstep. During a single computation, all incoming messages to the vertex for that superstep must be handled, and possibly all outgoing edges must be iterated over to send messages to neighbouring vertices. Iteration is a weak spot for parallel systems, as it is not executed in parallel. Therefore, in order to cut on computation time, vertices should have as few edges as possible.

Chapter 4

Traffic in a graph

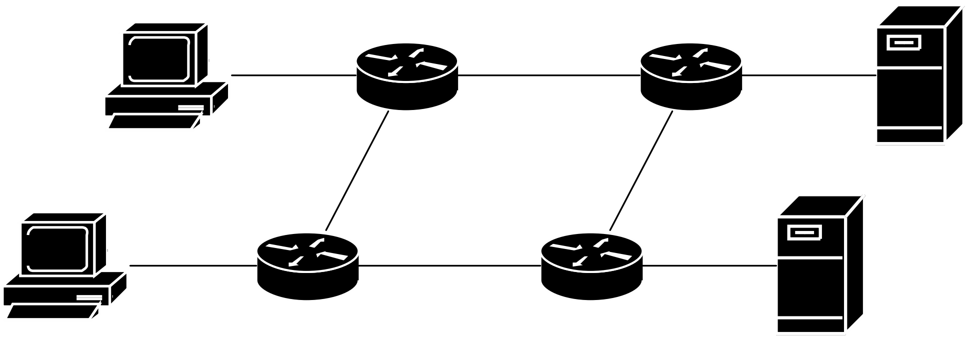
The internet is a world-wide network where all hosts can communicate with each other. Seen from a consumers' viewpoint, the internet is an outlet in the wall, or a wireless subscription. Connecting to the internet will give instant access to any website in the world (figure 4.1). Seen from the viewpoint of a network engineer, it is a vast infrastructure with countless routers connected with countless wires (figure 4.2). Connecting to the internet is to peer with an ISP, which peers with other ISPs, etc., making a global network where any host can communicate with virtually any other host. These two definitions are both correct, but they describe the internet on a different layer in the OSI model. The OSI model [26] is a model that partitions the internal functions of a communication system into abstraction layers.

Figure 4.1: A network observed on the transport layer; clients have direct contact to servers



The network layer describes how datagrams are routed between two hosts on the network. This delivery is realised by routers between these hosts, where each router forwards datagrams to a router that is closer to the destination. Datagrams are data packets, which start with an IP-header, which contains the source and destination IP address of the datagram. IP-addresses are fixed-length numbers of 32-bits (IPv4) or 128-bits (IPv6).

Figure 4.2: A network observed on the network layer; all devices are interconnected via routers



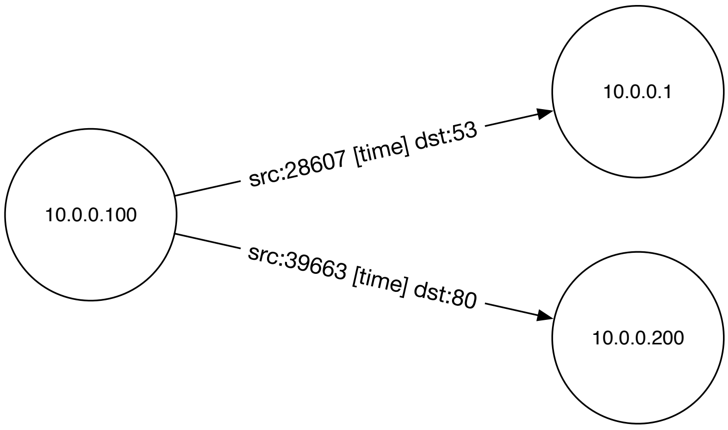
The transport layer describes segments being sent directly between hosts, not considering any routers. Typically, segments will have a protocol header, often TCP or UDP. The TCP and UDP headers contain two 16-bit port numbers (source port and destination port), which often indicate the kind of traffic in the packet. For example, port 80 indicates web traffic, 443 indicates encrypted web traffic, 53 indicates DNS traffic and 22 indicates SSH traffic.

4.1 Model optimisation

A possible approach to implement the NetFlow data model as a graph, is to represent IP addresses as vertices, and a combination of (1) ports and (2) time as edges (figure 4.3). This model contains vertices with many edges, which could take a long time to calculate per superstep. However, this model contains information that is not going to be used. Removing this information will allow computation to speed up, as unnecessary information will not take up computation time. In SpreadRank, the *ephemeral port number* is not important, so it can be discarded. Because SpreadRank will only look for spreading over identical port numbers (using port numbers as a metric for same-type communication), the vertices can be split. The result is a model in which a client that requests a DNS record and then a website will be represented by two vertices; one for the DNS client and one for the HTTP client. The model is inaccurate when compared to real-world TCP or UDP, in that the source port and the destination port are equal (whereas in the real world the connections initiated from the client would be initiated from an ephemeral port), but this model will make it easier to detect spreading over identical ports, as will be explained in chapter 7.

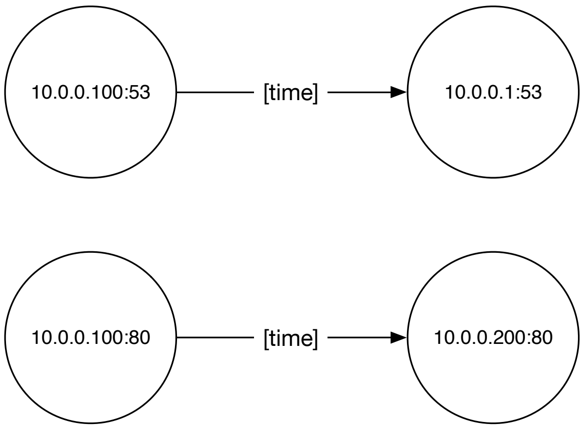
Figure 4.4 shows the optimised version of figure 4.3. There will not be any edge from a DNS vertex to a HTTP vertex. This will make it easier for Giraph to distribute the graph over different workers, in such a way that no vertices need to be

Figure 4.3: Graph with IP as vertex, ports and time as edge



moved in between supersteps.

Figure 4.4: Graph with IP and port as vertex, time as edge



4.1.1 Data conversion

In order to convert NetFlow to a graph in Giraph, an `InputFormat` for NetFlow data must be devised. Giraph has standard abstract readers and writers available that work with text formats. For easier implementation, a Bourne Again Shell (Bash) script was created that uses `nfdump` to convert NetFlow to Comma Separated Value (CSV) files, and a Giraph `EdgeInputFormat` was implemented that reads these CSV files.

The script filters its output so that only TCP and UDP flows remain where at least one port number is under 1024. It will invert flows where the source port is the lowest port. The script passes all its arguments to `nfdump`.

The recommended way to use this script is to convert NetFlow logs by day, so that multiple Giraph workers can read the graph at the same time. This will not only speed up reading, but it may prevent a worker crashing from reading a too large file, since the graph is stored in memory.

Using `nfdump` is not ideal, a better solution would have been to use a `NetFlowInputReader` in Giraph directly. At the time of writing, such a reader was not available. It is beyond the scope of this thesis to write such an input format reader.

```
#!/bin/bash
nfdump "$@" -o 'fmt:%ts %te %sa %da %pr %sp %dp %sas %das %in %out %tos %flg' | while read \
    flowstartdate \
    flowstarttime \
    flowenddate \
    flowendtime \
    srcaddr \
    dstaddr \
    proto \
    srcpt \
    dstpt \
    REST
do
    if [ "${proto}" = "TCP" -o "${proto}" = "UDP" ]
    then
        if [ "$srcpt" -lt 1024 -a "$dstpt" -ge 1024  -o  "$srcpt" -ge 1024 -a "$dstpt" -lt 1024 ]
        then
            [ $srcpt -lt $dstpt ] \
                && echo $flowstartdate\ $flowstarttime,$dstaddr,$srcaddr,$srcpt \\
                || echo $flowstartdate\ $flowstarttime,$srcaddr,$dstaddr,$dstpt \\

        fi
    fi
done
```

4.2 Spreading

Most hosts on the internet are either configured as a client or as a server, but some hosts are configured as both. A server is, for the purpose of this analysis, defined as a host that will receive connections. A client is, for the purpose of this analysis, defined as a host that will initiate connections. A connection is thus always initiated by a client, but it allows two-way communication, which allows client and server to exchange information.

This model is based on the assumption that most home users ("clients") will not run permanent services on their machines, but they will use e-mail and browse the web, which they do by contacting servers. In some cases, however, home users may choose to run servers on their own equipment as well. This can, for example, be a cheap solution to host a website or a mail server, or it can be done out of curiosity. On the other hand, servers may themselves be clients, either temporary, for example during maintenance, or as mode of operation, for example a slave DNS server.

In some cases when a client initiates a connection to a server, the server may relay this connection to another server, or it may open a similar connection to another server. This happens for example for proxy servers, which are configured to simply relay information. Another example is a DNS resolver, which may be queried for a record in the global domain name system. If it does not have this record available, it will query an authoritative server.

Spreading may also indicate the spreading of malware, for example a computer worm [19]. By looking at how far traffic spreads and which port number it uses, it is possible to get an insight of what happens in the network. This may prove helpful for finding infected machines in the network.

Chapter 5 Anomalies

This thesis will introduce the SpreadRank algorithm, which will detect traffic spreading on the transport layer. This means that it will detect hosts (clients and servers) that initiate the same kind of connections that they receive. The discriminator used to identify same-kind connections is the *contact port number* associated with the flow. The assumption is that a service both receiving and initiating connections to the same port is an anomaly. For example, a workstation with a web browser and e-mail client will initiate connections to ports 80 (HTTP), 443 (HTTPS), 587 (SMTP submission) and 143 or 993 (IMAP and IMAPS respectively). However, it is not expected that such a workstation would *receive* connections towards these port numbers. Neither is it expected that a web server would *initiate* connections directed towards port 80 or 443.

This is not true for all protocols. For example, a mail server can forward e-mail to another mail server via port 25 (SMTP), and this second mail server may forward it to a third mail server over port 25 and so forth. There are several other protocols that exhibit natural spreading, these are described in section 5.4.

5.1 Servers undergoing maintenance

Figure 5.1 shows spreading caused by a server undergoing maintenance. A server undergoing maintenance may connect to another server, for example to get the latest software updates. Additionally, there exists server software that includes a graphical user interface, which gives the server the same look and feel as a workstation. On these servers, the administrator may be inclined to use a web browser to download software, or even browse the web.

Figure 5.1: Observed spreading caused by a server under maintenance

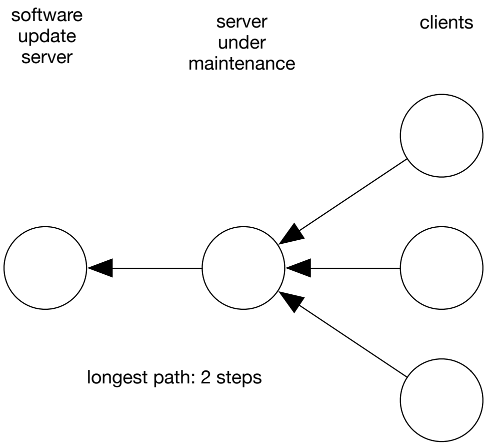
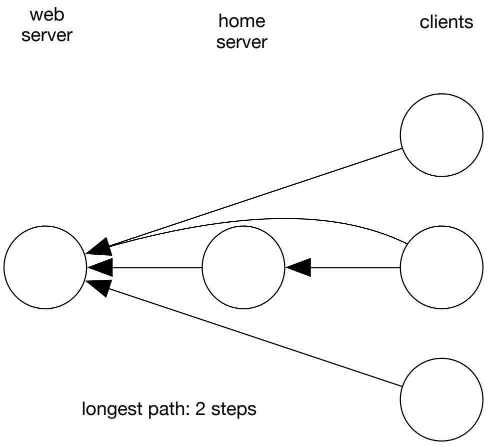


Figure 5.2: Observed spreading caused by a single home server

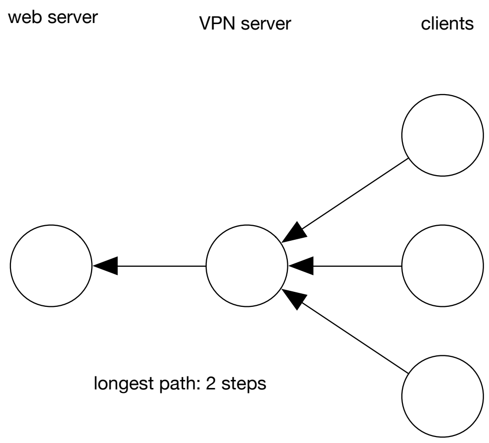


5.2 Home connections which run a personal server

Figure 5.2 shows spreading caused by servers run by home users on a shared public IP address. Users with a home server will often run the home server on the same IP address as the client. This is done either by simply using the same machine as server and client, or by using a NAT router, allowing a separate client and server machine to share a public IP address. When a client and a server share an IP address, this IP address will generally both receive and initiate connections. Since there is no direct way to distinguish whether incoming and outgoing flows are related, such a situation might register as a case of spreading. However, home servers are quite common, though a large trail of users with home servers contacting each other can appear to be as an anomaly.

5.3 VPN servers

Figure 5.3: Observed spreading caused by a VPN server



Clients can set up a secure connection to a VPN server (figure 5.3). All internet traffic initiated at the client (apart from the VPN traffic itself), will be sent to the VPN server in encapsulated form. The VPN server removes the encapsulating, and forwards the traffic towards the internet, mimicking the client. Thus, if the client uses another VPN service through this VPN server, the VPN server may appear to be a VPN client as well.

Some software allows using SSH, HTTP or HTTPS for VPN. Examples of software that can do this are sshttp [9] and Microsoft RRAS [23]. When a user connects to a VPN server over HTTPS, and then uses the VPN to connect to a HTTPS host, this can register as spreading. However, some VPN providers allocate dedicated public IP addresses to their clients, which means that connections are not forwarded over the same IP, which means that the VPN does not register as spreading.

5.4 Protocols with natural spreading

5.4.1 BGP

Border Gateway Protocol (BGP) is a protocol used by routers to exchange routing information. Even though routers are level three devices themselves, they exchange routing information with each other the same way hosts do. Routers both listen for incoming BGP connections, and send BGP to neighbouring routers at regular intervals. Since routers both receive and initiate BGP flows, BGP spreading is expected to be high.

5.4.2 DNS

DNS is used to find the IP address associated to a hostname. There are two types of DNS servers; *recursive* and *authoritative*. When a *recursive* DNS server receives a query, and does not have the requested record in its cache, it must forward the query to an upstream server, or find the authoritative DNS server for that record and forward the query there. An *authoritative* server has collections of records (zones) configured, though will only answer queries about these zones. Recursive DNS servers are expected to have high spreading, authoritative DNS servers are expected to have low spreading.

5.4.3 SMTP

Most e-mail clients are configured with a so-called "smart host", which is the server that all outgoing mail is sent to. This smart host will then find out which server handles e-mail for the recipient, and forward the message there. In some cases, the e-mail will pass through multiple servers before finally reaching a mailbox. Naturally, this means that some spreading in the SMTP protocol is expected.

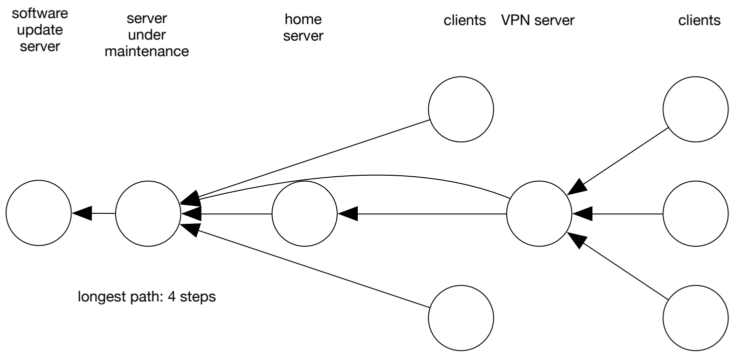
5.5 HTTP/HTTPS

A large number of different services use HTTP and HTTPS as underlying layer for their data. SpreadRank may therefore observe trails of machines connecting over port 80 or 443, without these flows actually being related. There is currently no direct way to view the difference between different services provided over HTTP or HTTPS, purely based on NetFlow data.

5.6 Combination

A combination of different occurrences of spreading may lead to larger spreading, as shown in figure 5.4; however, the longest path is still relatively short, which means that relatively short paths are not anomalies. Longer paths will typically be anomalies, though it depends on the protocol how long the path must be before it is considered an anomaly, for instance, one would expect higher spreading in SMTP and DNS than in HTTP.

Figure 5.4: Observed spreading caused by a combination of benign factors



5.7 Worm infection

A worm is a piece of software that is programmed to infect other hosts and start instances of itself on these hosts. It infects hosts by looking for systems that run unpatched software, abusing a known security hole to get access to the service and install itself on the host. After infection, the victim host will also try to infect other hosts [19]. This process can go on for ever, until a system administrator starts blocking these flows. Consequently, a worm that successfully spreads over many hosts, will show large spreading.

5.8 Peer to peer traffic

Peer to peer traffic is another case of spreading that happens intentionally; a hallmark example is BitTorrent. BitTorrent is a protocol for distributed file sharing, where all clients will make chunks they have downloaded available to other clients. This differs from the traditional client/server model, where clients do not spread information to other clients but get all their information from a central server. BitTorrent, however, operates over many different port numbers, which makes it difficult to match incoming and outgoing flows together.

The same is true for Tor traffic. Tor is a mechanism for anonymous internet access, which works by encrypting traffic and routing it at random through multiple hops, before sending it to the final destination. Tor also operates on random ports, which makes it difficult to match flows together.

5.8.1 Stealth worm

Due to BitTorrent and Tor being able to hide their activity from an algorithm such as SpreadRank, the question arises whether a worm can do the same thing to hide its activity. The important difference between benign traffic, such as BitTorrent and Tor, and malicious traffic such as worms, is consent from the owner of the host. If a host is participating in BitTorrent or Tor, it is because the owner of this host voluntarily installed software to supports these protocols. Because these protocols can rely on software being installed, they can use virtually any connection model imaginable.

This is different for malware; the initial infection must happen through a known vulnerability in the host. This vulnerability will typically be part of the operating system or part of installed software. Malware has therefore limited possibilities for infection, and must use the port of the service it wants to attack. Thus, in order to attack a specific service, malware must target the same port for every attack.

Once a host has been infected, the malware can install itself, and it may communicate with command-and-control servers through stealth communication channels that may not be as easy to detect; however, it will in all likelihood still try to attack other hosts, to increase the amount of infected hosts, for which it will have to generate observable traffic, for reasons mentioned earlier.

Chapter 6 DOSRank

To test Giraph, two simple algorithms, DOSRank and ReverseDOSRank are created, which count the amount of respectively incoming and outgoing edges from or to a vertex. This proof of concept will count the amount of flows per service. It can be used to detect if a machine is executing or suffering from a Denial of Service (DoS) attack, as these are typically identified by a large number of flows. These two algorithms will count respectively the amount of outgoing and the amount of incoming edges. DOSRank (6.1), which will calculate outgoing edges (incoming connections) is trivial; it will set its value during the first superstep and then vote to halt. ReverseDOSRank (6.2) is a bit more complex; since incoming edges are not visible in Giraph, the first superstep is used to send a message over every edge. During the second superstep, every vertex will receive an amount of messages that is equal to its amount of incoming edges.

$$f(x) = N^{-}(x). \tag{6.1}$$

$$f(y) = N^{+}(y). \tag{6.2}$$

Algorithm 6.1 DOSRank

```
result = vertex.getNumEdges();
vertex.voteToHalt();
```

6.1 Expected results

The algorithm will find IP addresses that are associated with large amounts of flows. Opening a large amount of flows is different from sending large amounts of data, as sending a large amount of data is often a completely legitimate thing to do: For example, sharing files via peer-to-peer protocols or serving large files over HTTP. A

Algorithm 6.2 ReverseDOSRank

```
if superstep = 0 then
begin
    result = 0;
    sendMessageToAllEdges();
end
else
begin
    while(messages.hasNext)
    do
        message.next();
        result++;
    end;
    vertex.voteToHalt();
end.
```

large amount of flows; however, may be an indication that something is amiss. It can indicate port scanning, or even targeted attacks like SYN flooding; however, a large amount of flows can also simply indicate a very active or popular host.

6.2 Purpose

These algorithms are just a proof-of-concept to show that graph systems can be used for traffic analysis. The algorithm itself is most likely easier to implement using other systems. A simple MapReduce algorithm could probably yield the same results a smaller amount of time [15]; however, this algorithm does show that Giraph can be used for traffic analysis, and it opens the door for more complex algorithms, like SpreadRank.

Chapter 7 SpreadRank

The previous algorithm, DOSRank, calculated the amount of flows. While a large amount of flows may indicate that something is wrong, it may also simply indicate a very active node. This chapter proposes SpreadRank, an algorithm to find hosts that are standing out due to the spreading they are causing. SpreadRank calculates how far connections spread. Spreading as an anomaly is described in chapter 5.

7.1 Overview of algorithm

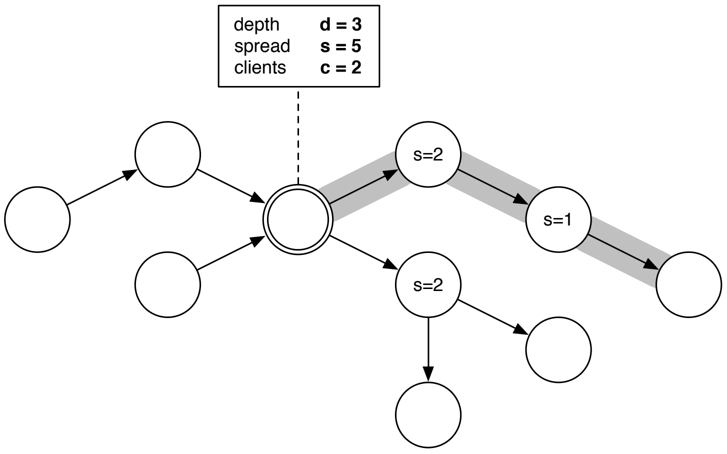
This thesis proposes SpreadRank, an algorithm to determine how far similar traffic spreads over hosts. *depth* in SpreadRank is defined in equation 7.1 and illustrated in figure 7.1. The value of d is the longest path from a host initiating a connection to another host receiving a similar connection. *spreading* in SpreadRank is defined in equation 7.2 and illustrated in figure 7.1. The value of s is the sum of all neighbours' d value, plus the amount of neighbours. Another way of looking at SpreadRank, is that during the first superstep, all nodes will send a token in the opposite direction of the flow. This token contains the timestamp from the flow. During all subsequent steps, all nodes that have received a token will send tokens again, but *only to flows that have a lower timestamp than the token. Depth* is then defined as the last superstep during which tokens were received, and *spread* is defined as the total amount of tokens received.

$$d(x) = 1 + \max_{y \in N^-(x)} d(y). \tag{7.1}$$

$$s(x) = \sum_{y \in N^-(x)} d(y) + \left| N^-(x) \right|. \tag{7.2}$$

Where E is the set of all edges in the graph, $f(x)$ and $f(y)$ are vertex values and $N^-(x)$ is a set of all incoming edges from vertex x.

Figure 7.1: Small example graph with results of one node, all nodes are hosts



SpreadRank bears similarities to Google's PageRank [17]. Both algorithms use a graph model where vertices and edges represent real-world elements. PageRank's model consists of web pages and hyperlinks and SpreadRank's model consists of services and flows. Both algorithms calculate values for vertices based on how they are connected to other vertices. The algorithms do, however, calculate different things, and as such use different metrics for calculating vertex values. This is most apparent in that in PageRank vertices send each other part of their current score, while in SpreadRank vertices simply send the timestamp of a flow start. This is because time is not important in PageRank; it does not matter who made a link first, while for SpreadRank it is important to know when a flow occurred, as spreading must occur in chronological order.

7.2 OSI model

The network layer provides stateless communication between hosts, and can be used to observe which hosts communicate with each other by means of IP addresses. Additionally, the IP header contains a protocol number, which gives some rough idea of the type of service. The transport layer (layer 4) provides end-to-end communication, and can be used to observe what kind of services are being used, by looking at the protocol header. Internet traffic consists mostly of TCP traffic, which makes TCP port numbers an interesting thing to look at. SpreadRank only looks at TCP traffic, and assumes that identical port numbers indicate identical types of service.

SpreadRank must not be confused with the Dijkstra algorithm [6], which can be used by layer 3 devices to find the shortest path between hosts. SpreadRank operates on NetFlow data, which contains IP addresses of hosts and port numbers, which is layer 4 data. The path that is observed is not a path via internet routers, but a path of hosts imitating each others behaviour.

7.3 Data types

In the SpreadRank algorithm proposed in this thesis, IP-address and port number tuples are represented by vertices (figure 7.2). The choice for this combination is due to spreading occurring over the same port, which is the best NetFlow metric available to indicate type of traffic. The vertex's value (figure 7.4) contains how far the traffic reaches (depth) and how far their traffic spreads (spreading). Flows are represented by edges. The edge has the flow start timestamp as value (figure 7.3).

Figure 7.2: Data type of a SpreadRank vertex ID

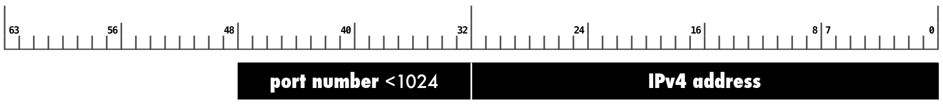


Figure 7.3: Data type of a SpreadRank edge

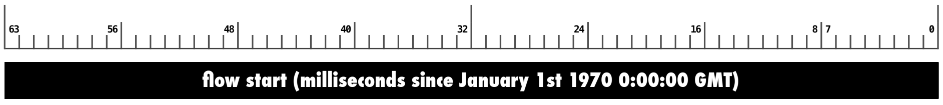
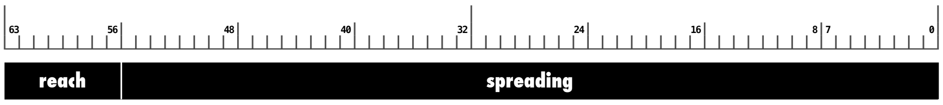

63
56
48
40
32
24
16
8
7
0
flow start (milliseconds since January 1st 1970 0:00:00 GMT)

Figure 7.4: Data type of a SpreadRank vertex value



7.3.1 Filtering

Before calculation, all flows that do not represent a TCP or UDP connection are removed, as well as every flow not between a contact port and an ephemeral port. Since the difference between a contact port and an ephemeral port may be vague, a contact port is assumed to be a well-known port, which is any port number below 1024 [20]. Any port from 1024 and up is assumed to be an ephemeral port. Different operating systems use different port numbers as ephemeral ports, but no operating system uses ports under 1024 as ephemeral ports (section 4.1.1).

Additionally, all double flows are removed, which means that there are always zero, one or two flows between two edges (no flows, one-directional, bi-directional).

For every vertex-pair, for every direction, the oldest edge is retained. Since each edge has a timestamp as value, the edge with the lowest value is retained and edges with higher values are removed.

7.3.2 Loop protection

The removal of redundant edges is to provide a loop protection mechanism. In a graph system, it is difficult to detect loops; During calculation, only information about the vertex itself is known (outgoing edges, vertex id, vertex value, incoming messages). It is not viable to include a trail of previous edges to the messages that are exchanged after the calculation; this would be expensive in terms of memory.

In the specific case of SpreadRank, this problem can be solved by only considering the first time a flow has been observed, and to ignore all subsequent occurrences (figure 7.5). If a host initiated a type of connection before it ever received it, it certainly did not initiate these connections because of the connection it received, and it is not a case of spreading (figure 7.6). Multiple hosts can initiate a connection before spreading occurs, in which case it is assumed that both are responsible (figure 7.7 and 7.8). It is possible that spreading is incorrectly assumed due to the log not observing a previous connection (figure 7.9). This can for example happen when logging started after the second host initiated a connection, but before the first host did.

Figure 7.5: Removal of duplicate flows

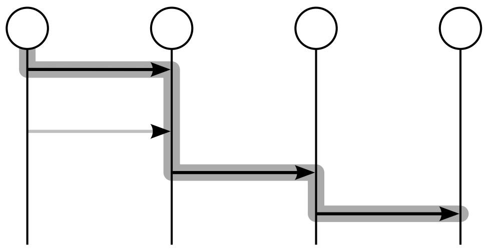


Figure 7.6: Spreading can only occur in sequence

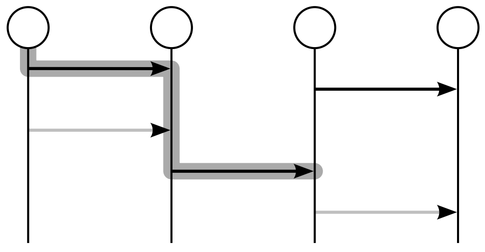


Figure 7.7: Spreading with two clients and one server

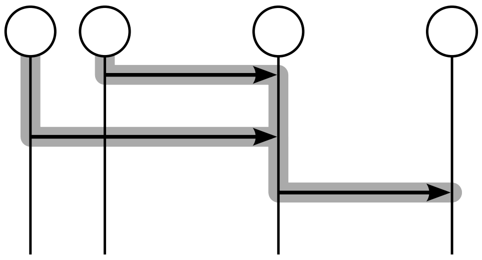


Figure 7.8: Spreading with two clients and two servers

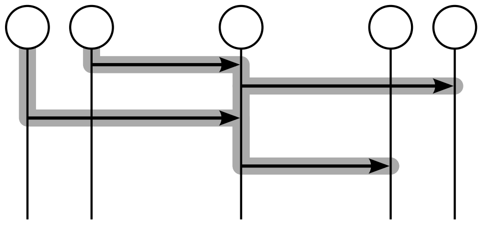
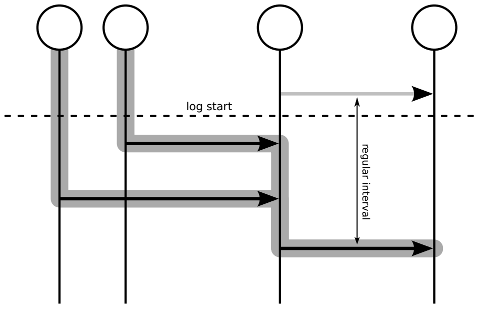


Figure 7.9: False positive spreading



$$S = T_{\text{firstReceived}} < T_{\text{firstSent}} \tag{7.3}$$

7.4 Implementation

In Giraph, it is not possible for a vertex to enumerate its incoming vertices, only outgoing vertices are available. For this reason, all direction edges are *reversed* in the model, so that they point towards the *initiator* of the traffic, not to the *receiver*. During the first superstep, all vertices will send a message over all outgoing edges (this message goes in the opposite direction of the original flow). The message consists of the value of the edge, which is the time that the connection was attempted (section 7.3). During all subsequent supersteps, all vertices that receive at least one message, will send one message over every outging edge *that has a lower value than the highest value of any message received.* The reason for this is that spreading must happen sequentally; when a host X sends to Y and then receives from Z, it is not a case of spreading. it is a case of spreading when host X receives from Z and then sends to Y.

Algorithm 7.1 SpreadRank

```
if superstep = 0 then
    removeDoubleEdges();
    for edge : outgoingEdges
    do
        edge.send(edge.value);
    end
end
else
begin
    lastMessage = max(messages);
    for edge : outgoingEdges
    do
        if edge.value < lastMessage
        then
        begin
            edge.send(edge.value);
        end
    end
end
vertex.voteToHalt();
```

7.5 Expected results

7.5.1 Depth

The amount of supersteps is expected to be relatively low, except for protocols that naturally exhibit spreading. A high amount of supersteps indicates that the host has generated a type of network traffic that is also generated by the hosts it has sent this traffic to, and that this process has repeated itself over several unique hosts, forming a path.

7.5.2 Spreading

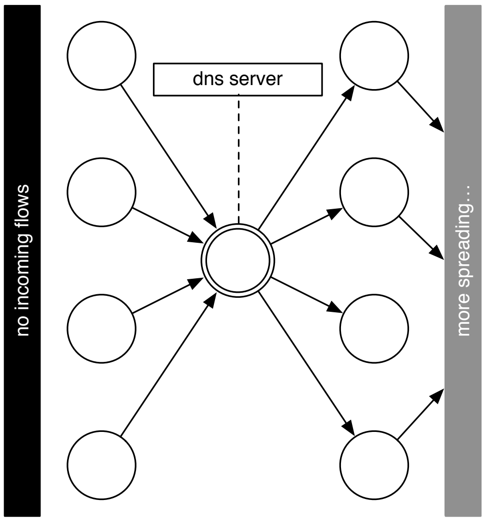
A high amount of incoming messages indicates that this host has initiated many connections. This is typical behaviour for a client, and for protocols that have natural spreading (for example DNS resolvers and BGP). Hosts that show high depth (large amount of supersteps) but have a relatively small spreading, may be part of some internal group, where information spreads between members of the group, but not outwards.

7.5.3 Clients

The amount of clients is simply a count of the incoming edges, similar to DOSRank. Since all edges are reversed (section 7.4), the amount of clients can simply be calculated by counting a vertex's outgoing edges in Giraph. In the SpreadRank implementation written for this thesis, the amount is not calculated during a superstep, but simply included in the output while the results are written to disk.

A typical case where this value is useful, is for finding DNS resolvers. Users (clients) of the DNS resolvers will typically have a higher depth than the resolver itself, since the client initiated the connection (figure 7.10). These clients themselves should have zero incoming connections, otherwise they are resolvers.

Figure 7.10: Illustration of how a single DNS resolver will have many incoming and outgoing connections.



By having the possibility to exclude vertices with few clients from the end results, it is trivial to limit the results to vertices with the second-longest path. This will make it easier to focus on hosts that forward traffic, and not on hosts that only initiate traffic.

Chapter 8
Results

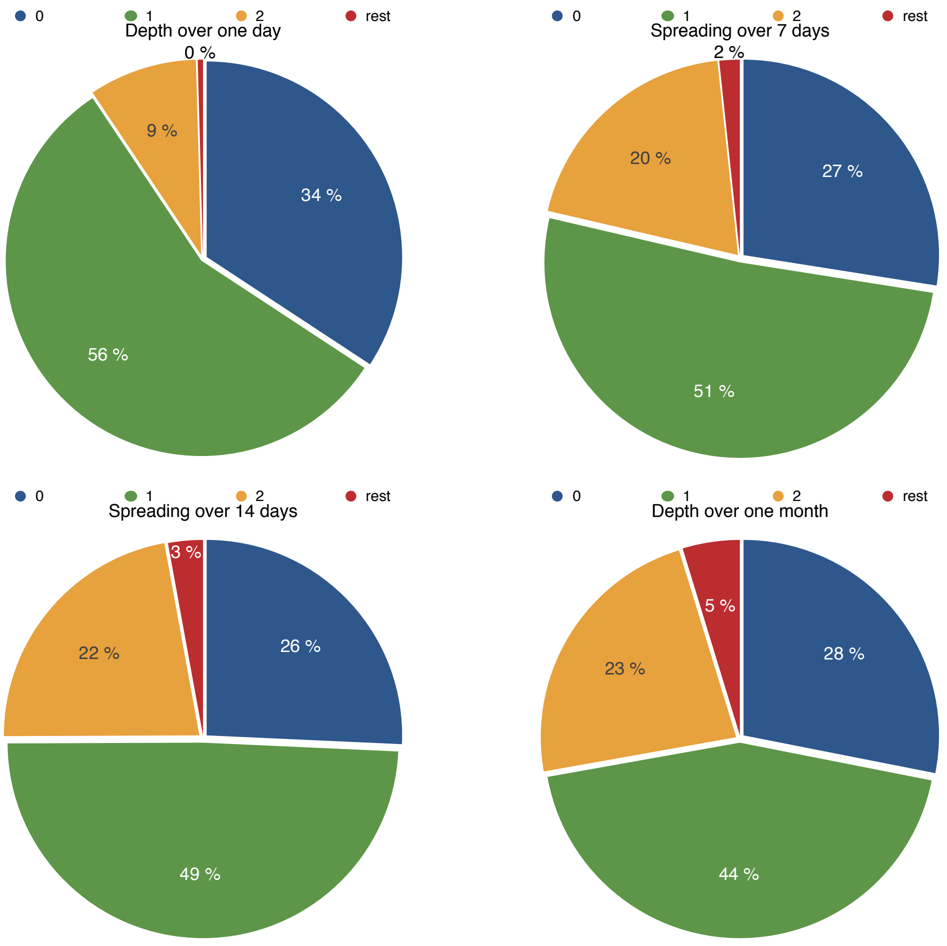
UNINETT has provided NetFlow data from one of their core routers, with flows from December 2013, in the NetFlow file format. To simplify implementation, `nfdump` was used to convert the flows to CSV files. SpreadRank was run on the flows logged by one router over one day, one week, two weeks and one month. Calculation of SpreadRank scores only took some minutes to complete, but the conversion to CSV took a few hours to complete. This was likely due to the fact that `nfdump` had to read the compressed NetFlow files from the Hadoop File System (HDFS), and then write uncompressed CSV files to the same HDFS volume.

After calculation, Giraph writes output files containing all vertex IDs as values. The vertex value (figure 7.4) consists of the depth and the spreading from the vertex. A Giraph `OutputFormat` was implemented in order to make the output files better parseable. The `OutputFormat` writes tab separated lines, containing *vertex ID*, *number of outgoing edges* (clients) and the vertex value (reach and spreading).

8.1 Longest path

Most services on the internet do not exhibit spreading. Figure 8.1 shows a plot of spreading per vertex, calculated over NetFlow information from one router over one day and over one month. Most vertices have a spreading of zero, one or two. A spreading of zero typically indicates that the host is a server, a spreading of one typically indicates that the host is a client (its traffic reaches the servers) and a spreading of two often indicates a hybrid, though it can also indicate a proxy server. Higher spreading indicates hosts that do participate in spreading. This is typical for DNS servers, SMTP servers and BGP routers.

Figure 8.1: Spreading per vertex

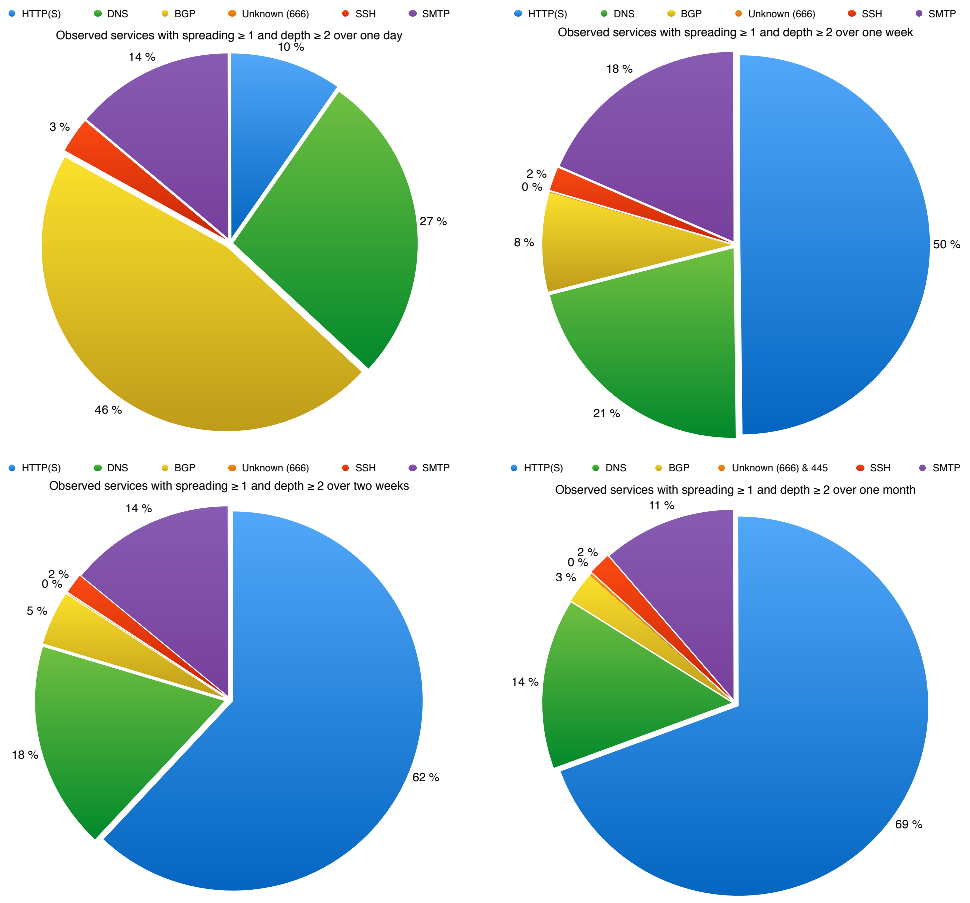


8.2 Protocols

After filtering traffic such that only services with a spreading of one or higher, and a depth of two or higher remain, only a few protocols show up (figure 8.2). This confirms the assumptions stated in section 7.5.1 and chapter 5, which is that only certain protocols exhibit spreading.

BGP, Domain Name System (DNS) and Simple Mail Transfer Protocol (SMTP) are expected to spread, due to the way the protocols work. BGP is used between routers to calculate shortest paths; routers need to keep each other updated and will therefore initiate connections to each other at regular intervals. DNS and SMTP are both protocols where clients send their request to a server that is close to them, and the server will handle the request itself, or forward to the request to another server. In the case of DNS, this means that the server will answer with information from its cache, or look the requested record up itself. In the case of SMTP, this means that the server will forward e-mail to the mail server of the recipient. These services yield high depths, as virtually any server providing these services will eventually forward requests it receives.

Figure 8.2: Distribution of protocols that exhibit spreading



The Secure Shell (SSH) and HyperText Transfer Protocol (HTTP) protocols also exhibit higher spreading. The reason for this may be that home servers often serve web pages, and Linux-based home servers will typically also accept SSH connections for remote management. HTTP spreading is discussed in more detail in section 5.5.

Spreading is also observed on ports 666 and 445. Port 666 is officially allocated as the port for both the Multiple Device Queueing System, and computer game

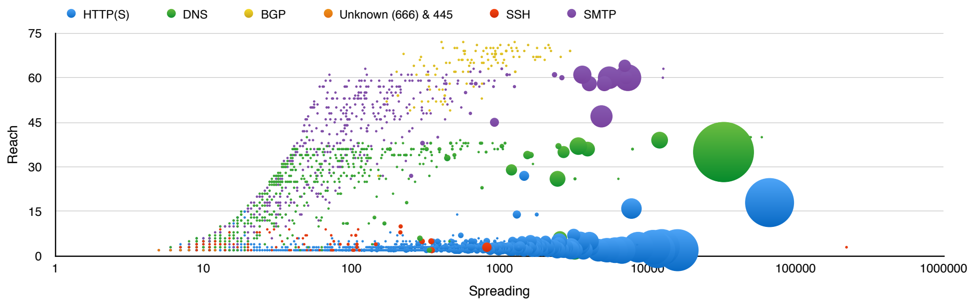
Doom [20], but in practice it is also used by viruses. Only one instance of spreading over port 666 was observed over one month, but due to the ambiguity of the port, it is not clear what kind of traffic this was. Port 445 is used by Microsoft Windows for sharing files, but it is also abused for denial of service attacks [12]. Because of this, it is unknown whether this spreading was due to an attack, or simply some users that had used Microsoft Windows File Sharing.

8.3 Diagram

Individual observed services with a reach of one or higher are plotted in figure 8.3. It is based on one month's worth of NetFlow.

- The X axis shows spreading – the amount of hosts reached via spreading – the amount of messages received in total.
- The Y axis shows depth – the longest path.
- The size of the dot shows the amount of observed clients – the amount of incoming connections to the host.

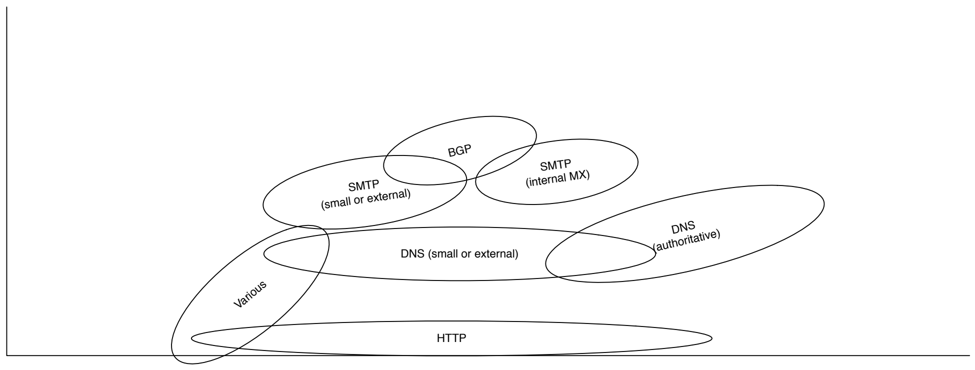
Figure 8.3: End-hosts scored with SpreadRank over one month



8.4 Observed anomalies

Figure 8.3 shows all observed IP-address + port number pairs with a depth of 2 or higher, during one month. Similar diagrams from different timespans can be found in appendix B. The best way to observe anomalies in this diagram, is by Multi-class Anomaly Detection. Multi-class classification assumes that the data set contains different distinct types of data [3]. In the case of SpreadRank, these types of data are the port numbers. Different ports are indicated with different colours in the diagram, and it is apparent that there is a pattern (figure 8.4). It apparent that there are some outliers, but there is no clear border between normal observations and anomalous observations; however, a simple visual observation of the diagram does give some interesting results.

Figure 8.4: SpreadRank pattern over one month



8.4.1 Botnet

Figure 8.5: HTTP and SSH traffic in SpreadRank over one month

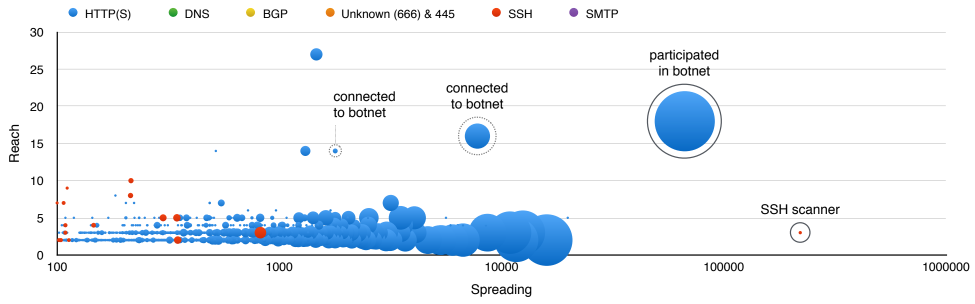


Figure 8.5 shows HTTP and SSH services. Dots marked with a solid line are linked to hosts that were reported to CERT. The tiny red SSH node at the right hand side has been involved in SSH scanning, where it connects to many IP addresses in the hope to find an SSH server with a password it can guess.

The large HTTP node at the right hand side is a host at NTNU, and was confirmed participating in a botnet during December 2013, but was not discovered until January. The host had many incoming and outgoing connections over port 80 in December. After analysing the logs with more detail, it became apparent that it was contacted very often by servers hosting questionable content. These servers were contacted by various other clients at NTNU at a regular interval, but starting the 24th of December, some of these servers started contacting the host at NTNU.

On January 1st, 2014, UNINETT CERT received a report stating that the machine participated in a botnet.

By matching the IP addresses that this botnet host had contact with, two additional hosts (marked with dotted lines) were discovered participating in the botnet. Apparently this botnet does not make use of stealth technologies as outlined in section 5.8.1. A possible reason for this, is that port 80 is a relatively "safe" port to communicate over; due to the high usage of HTTP ports for many different types of traffic (chapter 5), system administrators may be inclined not to consider HTTP traffic when looking for anomalies. SpreadRank has therefore proven to be a useful tool for finding these kinds of anomalies.

The two hosts (marked with dotted lines) were not reported to CERT, and were therefore never investigated. According to information from DNS, the host that got reported to CERT is a machine at a faculty at NTNU. The two hosts that were found using SpreadRank are computers at two different student villages at NTNU. More precise information is available at UNINETT, but cannot be published in this thesis for privacy reasons.

8.4.2 DNS

Figure 8.6: DNS traffic in SpreadRank over one month

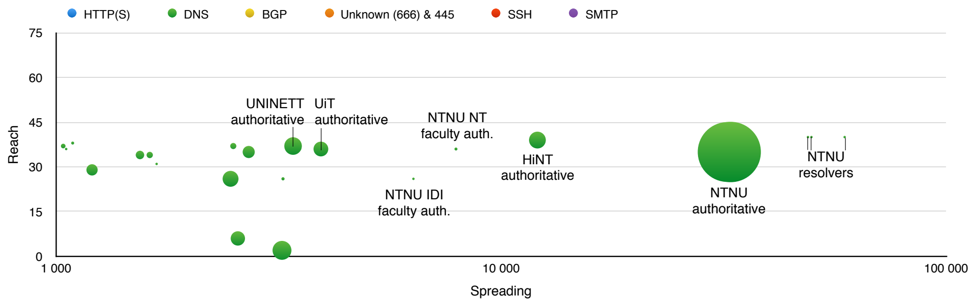


Figure 8.6 shows DNS services, marked by which institution manages them. DNS resolvers are used to look up DNS records for hosts inside the network. These resolvers cannot be used from the outside, which is why the dots are small; the clients are inside the institution, and are therefore not logged by UNINETT's core routers (figure 2.1).

Authoritative DNS servers that cause spreading may seem like an anomaly; such authoritative DNS servers answer queries from clients and do not send queries themselves; however, there is an explanation for this spreading: In order to keep DNS zones available, most domains use multiple DNS servers. These DNS servers must keep their records information synchronised, and will therefore periodically contact a master server if it is not itself the master for the DNS zone.

8.4.3 SMTP

SMTP is the protocol used for mail delivery, where sometimes an e-mail is forwarded a couple of times before it reaches a mailbox. This causes spreading, which can be seen in figure 8.3. There are some SMTP servers that exhibit very high spreading, but these are simply hosts that handle large amounts of mail. The diagram with SMTP servers would give a very comprehensive overview of e-mail usage at NTNU, but for privacy reasons this diagram is not included. Such a diagram would give information about which parties NTNU communicates with most, which includes companies that recruit at NTNU.

8.5 Performance

UNINETT made a server cluster available for the analysis. This cluster is installed with Apache Hadoop [2], and consists of 15 worker machines with 4 CPUs and 16 GB RAM each. In addition, there are two HDFS name-nodes and one YARN manager, which have 4 CPUs and 10 GB RAM each.

Figure 8.7 shows how SpreadRank performs on this cluster, by NetFlow logs over one day, one week, two weeks and one month. The logs are cumulative, so the one week log also contains the information from the one day log, etc. Since every task is divided over multiple workers, of which multiple can run on one worker machine, the calculation times may fluctuate. The figure shows one measurement where SpreadRank is run four times, for different NetFlow log sizes. An interesting result is that the calculation of one week takes slightly longer than the calculation of two weeks. This is probably by accident, the job may have been scheduled in such a way that multiple workers were located on the same physical machine.

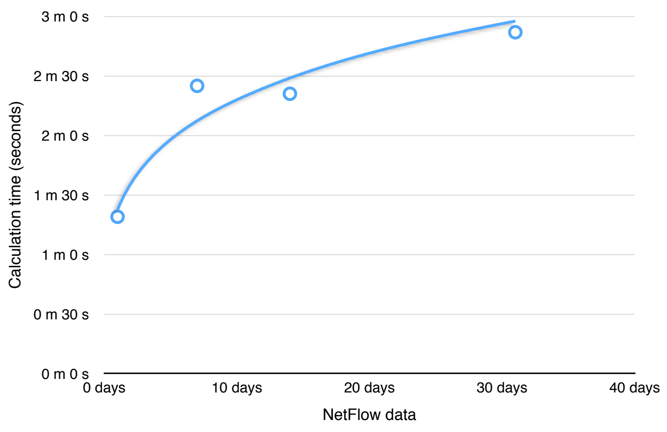
8.6 Comparison to other systems

Software already exists to handle NetFlow logs. The goal of most of this software is to make NetFlow logs more easily readable. UNINETT currently uses NFSEN[1] for analysis of NetFlow information. NFSEN provides a web interface from which the NetFlow logs are easy accessible as text or aggregated in a diagram (figure 8.8). The aggregated data from NFSEN can also be generated using MapReduce [15].

SpreadRank, as implemented in this thesis, does not provide any database storage, diagram generation or user interfaces. The plots shown in this thesis were created

[1]http://nfsen.sourceforge.net

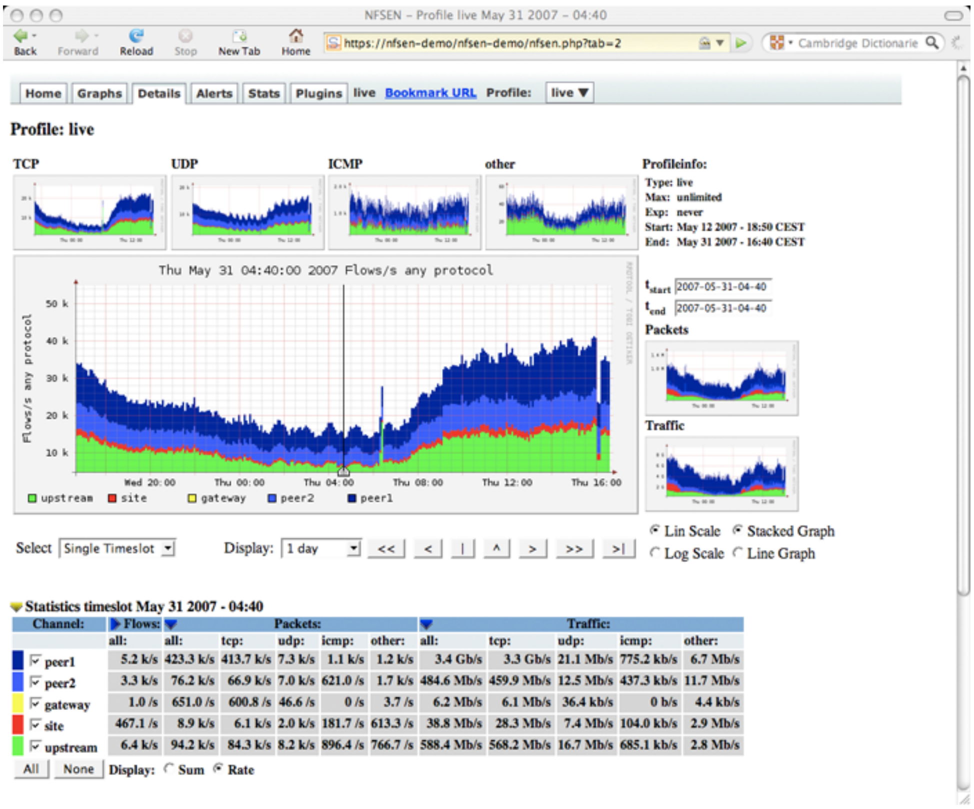
Figure 8.7: Time used to calculate SpreadRank



using modeling tools and spreadsheet software. SpreadRank, instead, aggregates flows in a more recursive way, which gives additional metrics about hosts. These metrics would be useful to integrate in NFSEN, but NFSEN in its current form does not support plugins. Software similar to NFSEN is FlowViewer[2] and SiLK [3].

[2]http://sourceforge.net/projects/flowviewer/
[3]https://tools.netsa.cert.org/silk/

Figure 8.8: Screenshot of NFSEN (http://nfsen.sourceforge.net/details-graphs.png)



Profile: live

Profileinfo:
Type: live
Max: unlimited
Exp: never
Start: May 12 2007 - 18:50 CEST
End: May 31 2007 - 16:40 CEST

t_{start} 2007-05-31-04-40
t_{end} 2007-05-31-04-40

Statistics timeslot May 31 2007 - 04:40

Channel:	Flows:	Packets:					Traffic:				
	all:	all:	tcp:	udp:	icmp:	other:	all:	tcp:	udp:	icmp:	other:
peer1	5.2 k/s	423.3 k/s	413.7 k/s	7.3 k/s	1.1 k/s	1.2 k/s	3.4 Gb/s	3.3 Gb/s	21.1 Mb/s	775.2 kb/s	6.7 Mb/s
peer2	3.3 k/s	76.2 k/s	66.9 k/s	7.0 k/s	621.0 /s	1.7 k/s	484.6 Mb/s	459.9 Mb/s	12.5 Mb/s	437.3 kb/s	11.7 Mb/s
gateway	1.0 /s	651.0 /s	600.8 /s	46.6 /s	0 /s	3.7 /s	6.2 Mb/s	6.1 Mb/s	36.4 kb/s	0 b/s	4.4 kb/s
site	467.1 /s	8.9 k/s	6.1 k/s	2.0 k/s	181.7 /s	613.3 /s	38.8 Mb/s	28.3 Mb/s	7.4 Mb/s	104.0 kb/s	2.9 Mb/s
upstream	6.4 k/s	94.2 k/s	84.3 k/s	8.2 k/s	896.4 /s	766.7 /s	588.4 Mb/s	568.2 Mb/s	16.7 Mb/s	685.1 kb/s	2.8 Mb/s

Display: Sum [✓ Rate]

Chapter 9

Conclusion

This thesis has stated the current practices in traffic anomaly detection. Current traffic anomaly detection is based on aggregating flows, or by identifying high-intensity traffic. The thesis defines the concept of spreading, which describes the phenomenon of a host initiating the same kind of connections it receives, where same-kind refers to usage of the same TCP/UDP ports. It is argued that spreading is uncommon for most protocols on the internet today, which makes it an anomaly.

The usage of graph systems is proposed as a means to measure spreading. Multiple graph systems are available today, this thesis uses the Giraph system. Using Giraph, NetFlow information provided by UNINETT is converted to a graph, where spreading is calculated using SpreadRank, an algorithm introduced in this thesis.

SpreadRank works by making a graph of flow data. In this graph, vertices represent IP address and port number pairs (services), and edges represent flows, and have the flow start time as value. Every service is scored on its longest path towards another service (depth), and on how far it spreads its traffic (spreading).

An analysis of the results shows that only five percent of all *services* participate in *spreading*, and that for many hosts, their role can be determined by simply looking at their spreading. A spreading of zero typically indicates a server, one typically indicates a client (its traffic reaches to the servers) and a spreading of two often indicates a combination of both, though it can also indicate a simple proxy server.

Some protocols have natural spreading, for example BGP, DNS and SMTP; implementations of these protocols behave often both as server and client. SSH and HTTP are popular protocols which do not have natural spreading; however, these protocols exhibit spreading nevertheless. HTTP has a high spreading due to it being a protocol that is used for multiple purposes. An HTTP service may itself use HTTP to connect to another service. SSH allows users to “hop” from one SSH server to another.

SpreadRank has been successful in identifying spreading, and in doing so can be successfully used to find DNS resolvers, BGP routers and mail servers on the network. It has also been successful in finding hosts that participated in a botnet. It does so based on NetFlow data from core routers, without sending data into the network itself.

9.1 Future work

9.1.1 Automation

The current implementation of SpreadRank requires the analyst to execute many manual steps. A better `EdgeInputFormat` would reduce the amount of conversions needed, and speed up the overall process. The output could then be parsed to automatically find outliers.

9.1.2 Test-data

It was not known beforehand which attacks were present in the test-data. The experiment should be repeated with test-data with known attacks. The best time to do this, is most likely after a large worm outbreak.

9.1.3 Real-time monitoring

The experiment was conducted on static test data. For a real-life application, real-time monitoring is required, as this makes it possible to set automatic alarms when something is amiss. In order to work with real-time data, a sliding window is required, in which flows are added to the graph as they are observed, and old flows are removed. Giraph does not currently support this; however, systems that support this do exist, for example GraphX.

9.1.4 IPv6

The NetFlow data provided by UNINETT contains only flows between IPv4 hosts. The SpreadRank algorithm may need some modifications to be able to be used on IPv6. The two most important differences between IPv4 and IPv6 for SpreadRank are that IPv6 addresses are a lot longer and will therefore not fit in the current data types. Additionally, another traffic pattern will be observed regarding home servers. Where many home servers currently share their public IPv4 address with clients due to the use of technologies such as NAT, IPv6 makes it possible to run the server and client on different IPv6 addresses.

Additionally, the use of privacy extensions in IPv6 (a technology that lets clients randomise their IP address for anonymity) will prove to be both a challenge and an advantage for SpreadRank. It is a challenge because it will lead to more vertices; in the current model, every observed IP address + port number pair is a vertex, which means that a new vertex is created every time a client switches IP address. Randomised IP addresses are also an advantage, because it is not feasible to run a server on a randomised IP address. This may make it easier to designate vertices as clients at a very early stage, which reduces computation time.

References

[1] Ching Avery. Giraph: Large-scale graph processing infrastruction on hadoop. *Proceedings of Hadoop Summit. Santa Clara, USA:[sn]*, 2011.

[2] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[4] A Ching. Scaling apache giraph to a trillion edges. *Facebook Engineering blog*, 2013.

[5] Inc Cisco Systems. Understanding the netflow collector data file format. *Cisco NetFlow Collector User Guide.*

[6] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[7] Zhiqiang Gao and Nirwan Ansari. Differentiating malicious ddos attack traffic from normal tcp flows by proactive tests. *Communications Letters, IEEE*, 10(11):793–795, 2006.

[8] Wolfgang John and Tomas Olovsson. Detection of malicious traffic on back-bone links via packet header analysis. *Campus-Wide Information Systems*, 25(5):342–358, 2008.

[9] S. Krahmer. SSH/HTTP(S) multiplexer. Available online https://github.com/stealth/sshttp, 2014.

[10] Christian Kreibich, Andrew Warfield, Jon Crowcroft, Steven Hand, and Ian Pratt. Using packet symmetry to curtail malicious traffic. *ACM Hotnets-IV*, 200, 2005.

[11] Kun-chan Lan, Alefiya Hussain, and Debojyoti Dutta. Effect of malicious traffic on the network. In *Proc. of PAM*. Citeseer, 2003.

[12] Aleksandar Lazarevic, Levent Ertoz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. *Proc. SIAM*, 2003.

[13] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[15] Jan Tore Morken. Distributed netflow processing using the map-reduce model, 2010.

[16] Maziar Nekovee, Yamir Moreno, G Bianconi, and M Marsili. Theory of rumour spreading in complex social networks. *Physica A: Statistical Mechanics and its Applications*, 374(1):457–470, 2007.

[17] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[18] Janak J Parekh, Ke Wang, and Salvatore J Stolfo. Privacy-preserving payload-based correlation for accurate malicious traffic detection. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 99–106. ACM, 2006.

[19] Romualdo Pastor-Satorras and Alessandro Vespignani. Epidemic spreading in scale-free networks. *Physical review letters*, 86(14):3200, 2001.

[20] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700 (Historic), October 1994. Obsoleted by RFC 3232.

[21] Sherif Sakr. Processing large-scale graph data: A guide to current technology. *IBM Developerworks*, page 15, jun 2013.

[22] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[23] J. Vasudevan. How to configure the rras based vpn server to accept sstp connections. Available online http://blogs.technet.com/b/rrasblog/archive/2007/02/02/configuring-the-vpn-server-to-accept-sstp-connections.aspx, 2007.

[24] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[25] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[26] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.

Appendix A
Commands

During this study, SpreadRank was run on YARN, and results were written to text files. This appendix will show the commands used to convert NetFlow to CSV, start SpreaRank with these CSV files, and filter the results.

A.1 Convert NetFlow to CSV

NetFlow can be converted to a CSV file, with non-UDP and non-TCP flows filtered out, as well as flows between port numbers that are both over 1024. This conversion is done using FlowConvert.sh, though in order to generate separate CSV files, FlowConvert must be run for each time frame that a CSV file should span. This snippet of Bash code will run FlowConvert.sh 31 times, for every day of the month. The end result will be a directory named `trd_gw1_12_filtered.csv` with 31 files.

```
seq 1 9 | while read nr
do
    sh FlowConvert.sh -R trd_gw1/12/0$nr | cat > trd_gw1_12_filtered.csv/part-0000$nr &
done
seq 10 31 | while read nr
do
    sh FlowConvert.sh -R trd_gw1/12/$nr | cat > trd_gw1_12_filtered.csv/part-000$nr &
done
```

A.2 Execute SpreadRank

In order to execute SpreadRank on YARN, the application must be submitted to the YARN manager, so that it can deploy the SpreadRank jar file to all workers. The command takes as arguments the path of the jar file, and the fully qualified class-name of both Giraph. Giraph takes the fully qualified class-name of the SpreadRank computation class. Additionally, a format for reading the graph at the start, and writing the graph at the end must be provided. In this case, we use a directory to store the graph data. The directory contains chunks of the full graph.

```
yarn \
 jar giraph-rank-1.1.0-SNAPSHOT-for-hadoop-2.3.0-cdh5.0.1-jar-with-dependencies.jar \
 org.apache.giraph.GiraphRunner no.uninett.yorn.giraph.computation.SpreadRank \
 -eif no.uninett.yorn.giraph.format.io.NetflowCSVEdgeInputFormat \
 -eip /user/hdfs/trd_gw1_12_filtered.csv \
 -vof no.uninett.yorn.giraph.format.io.RankVertexOutputFormat \
 -op /user/hdfs/rank-out/IPSpreadRank_gw1_12 \
 -wc org.apache.giraph.worker.DefaultWorkerContext \
 -w 16 \
 -yj giraph-rank-1.1.0-SNAPSHOT-for-hadoop-2.3.0-cdh5.0.1-jar-with-dependencies.jar
```

A.3 Filter results

The raw results from SpreadRank, even though outputted by a custom `OutputFormat`, contain a lot of information that is not relevant. This simple Bash script will remove all vertices with a depth of 1 or less, and all vertices with no spreading or no clients. The program is built using these components:

- `cat`: concatenate all parts of the output
- `grep`: filter out all values equal to 0 or 1
- `sort`: make sure that the highest spreading is returned first
- `sed`: anonymize the results by removing IP addresses though keeping port numbers
- `less`: easier reading of the results in a terminal

Components can taken out if desired, to change the results. Total execution time of this command on one month worth of NetFlow data took about 10$\tilde{1}$5 seconds on UNINETT's server cluster.

```
cat rank-out/IPSpreadRank_gw1_12/part-m-* | egrep -vw '[01]' | sort -hrk 3 | sed 's/^.*://' | less
```

A.4 Aggregate

This Bash script can be used to determine how often a protocol engages in spreading. It will output two numbers per line, the second being the protocol number, the first being the amount of services observed. The program is built using these components:

- `cat`: concatenate all parts of the output
- `grep`: filter out all values equal to 0 or 1
- `sed`: remove IP addresses, otherwise the list would show all IP addresses with count 1
- `cut`: remove all but the first value (port number)
- `sort`: set the port numbers in order, required for `uniq`.
- `uniq`: aggregate the port numbers, and show count

Total execution time of this command on one month worth of NetFlow data took about 101̃5 seconds on UNINETT's server cluster.

```
cat rank-out/IPSpreadRank_gw1_12/part-m-* | egrep -vw '[01]' | sed 's/^.*://' | cut -f1 | sort | uniq -c
```

Appendix B
SpreadRank results

This appendix contains diagrams of SpreadRank being executed on NetFlow logs from one of UNINETT's core routers. These diagrams are made using different periods of NetFlow logs from the same core router.

Figure B.1: Vertices scored with SpreadRank using one day of flows

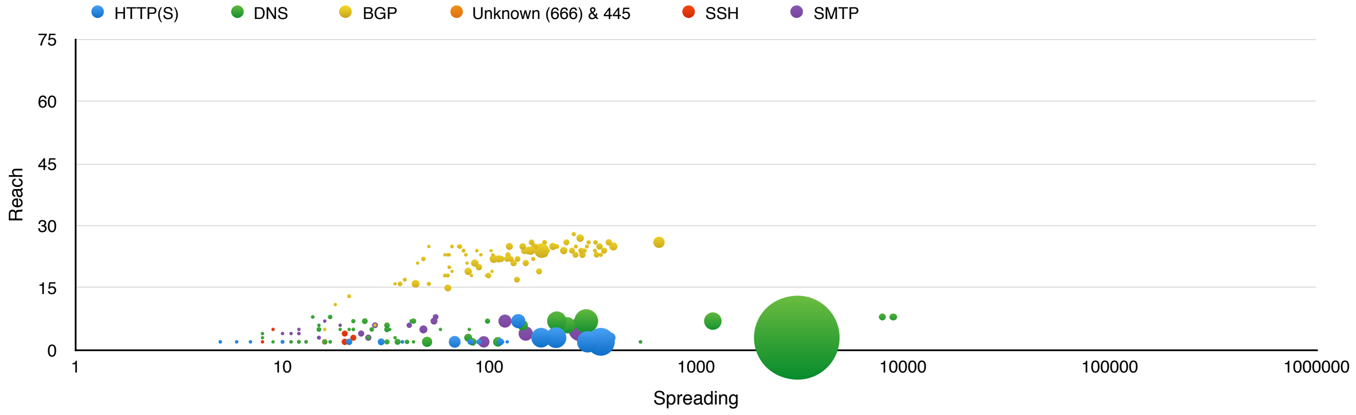


Figure B.2: Vertices scored with SpreadRank using one week of flows

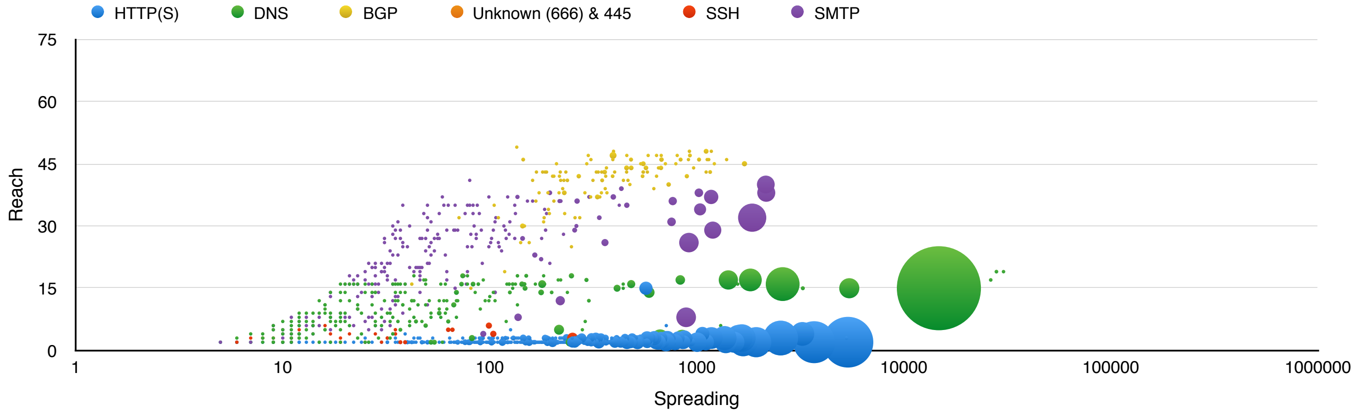


Figure B.3: Vertices scored with SpreadRank using two weeks of flows

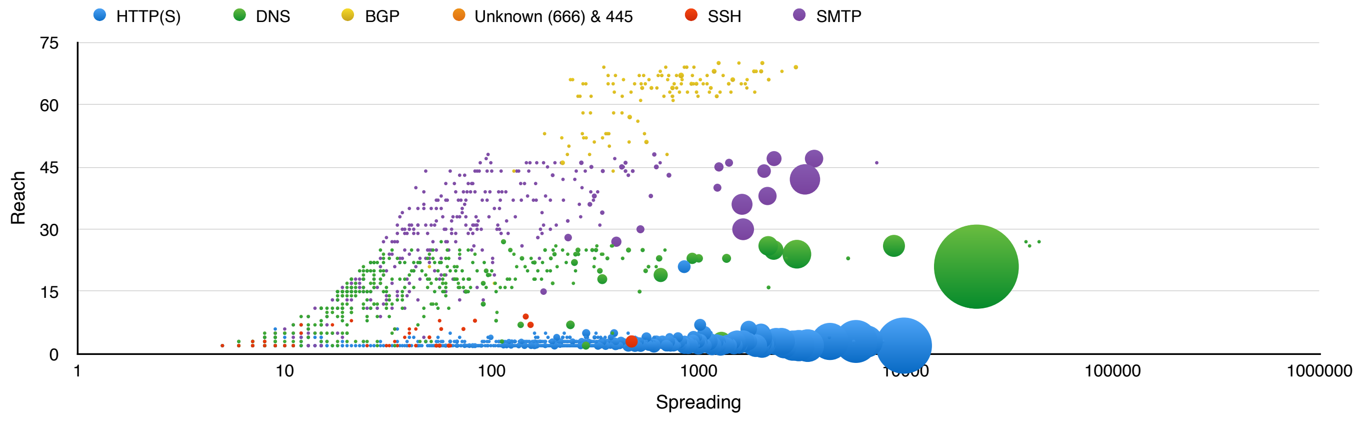
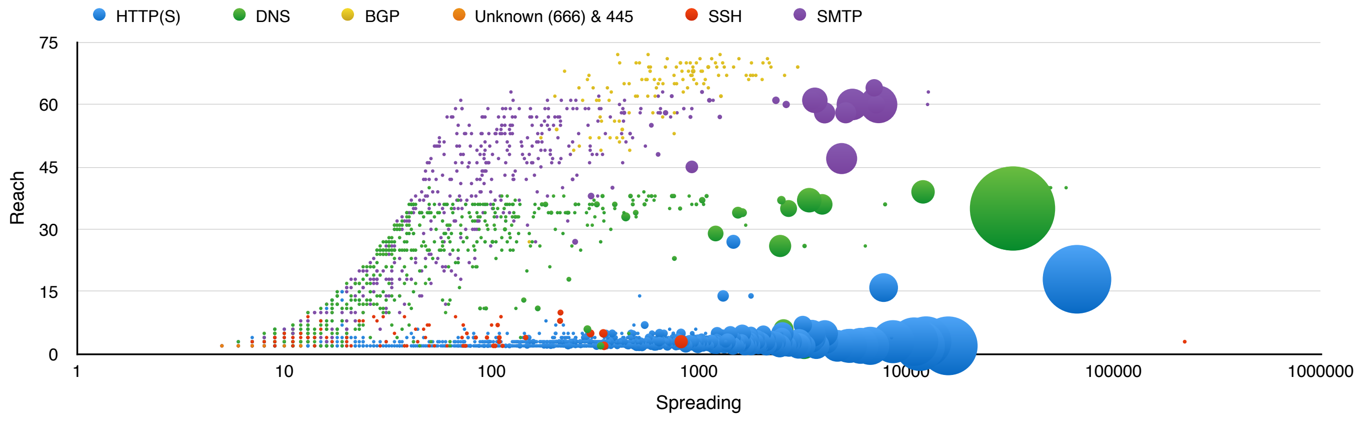


Figure B.4: Vertices scored with SpreadRank using one month of flows


HTTP(S)
DNS
BGP
Unknown (666) & 445
SSH
SMTP
Reach
Spreading