Improving Fidelity in Internet Simulation through Packet Injection

Conference Paper · September 2016

CITATIONS
0

READS
55

3 authors, including:

Barry Irwin
Noroff University College
183 PUBLICATIONS **877** CITATIONS
SEE PROFILE

Alan Herbert
Rhodes University
17 PUBLICATIONS **22** CITATIONS
SEE PROFILE

Improving Fidelity in Internet Simulation through Packet Injection

Craig Koorn[1*],Barry Irwin[2*], Alan Herbert[3*]
[*] *Security and Networks Research Group Department of Computer Science Rhodes University, Grahamstown, South Africa*
[1]craig.koorn@gmail.com,[2]b.irwin@ru.ac.za, [3]g09h1151@campus.ru.ac.za

***Abstract*—This paper describes the of extension implemented to the NKM Internet simulation system, which allows for the improved of injection of packet traffic at arbitrary nodes, and the replay of previously recorded streams. The latter function allows for the relatively easy implementation of Internet Background Radiation (IBR) within the simulated portion of the Internet. This feature thereby enhances the degree of realism of the simulation, and allows for certain pre-determined traffic, such as scanning activity, to be injected and observed by client systems connected to the simulator.**

I. Introduction

With the prominence of the Internet in modern times, the world has come to rely on the proliferation of the many networks and protocols which together form what is known today as the the Internet. The burgeoning demand for both new and existing Internet and network services has led to many organisations investing in these technologies; be it for the development of new technologies; or technologies in support of existing organisational functions [1]. For this reason, understanding these technologies forms a critical role within any such organisation, and often requires a substantial investment of capital and/or time.

Simulating these networks and their protocols allows for mitigating some of the costs associated with the development of new services, or software that makes use of these services. It allows for a better understanding of what is currently happening as well as what is likely to happen going forward. This is both applicable in Educational institutions where hands-on, practical experience is tantamount to an effective education; as well as in business contexts where the ramifications of going live without sufficient prior testing can be costly.

The need to purchase expensive hardware can, in some cases, often be omitted through instead utilising a software component which provides a practical means for testing and training. However, readily available software which meets these requirements at scale is in short supply. In lieu of this necessity, [2] presents NKM (Network Kernel Module) — a large-scale network routing simulator. Whilst NKM has proven effective in its ability to simulate vast and intricate network topologies, it still requires the presence of physical systems to provide meaningful interaction.

In this regard, NKM can be enhanced, so as to not only facilitate the process of simulating large networks, but to also cater to the simulation of the hosts which reside therein. This addition, coupled with its ability to provide a level of interaction previously unavailable within the system itself, may further assist in alleviating financial costs. Moreover, through representing these hosts virtually, the need to setup and assign dedicated hardware to this purpose diminishes. Not only does this assist in freeing up physical workspace, it too allows for representing these hosts in a manner which is easily modifiable and reproducible by other researchers.

While the addition of the lightweight virtualisation may assist in representing the nodes present in a network, it does allow for a realistic representation of the traffic charactarisitics on thee network itself. As the nature of the Internet is not one that is at all times passive and orderly, the fidelity of the system can be further be improved if a nore natural, live, representation of behavior is considered. This behavior can be simulated at an increased fidelity level through the injecting traffic which consists of datagrams, representative of the traits and desired characteristics once exhibited by a network at some point in time. Not only does this feature then add to the realism of the system, but it introduces new functionality, whereby this traffic, and the impact thereof, can be analysed and understood in a self-contained and isolated environment. Replay features also greatly improve the repeatability of experimental work.

This research aims to investigate the viability of these supporting a traffic injection and replay feature so as to create a well-rounded, realistic, feasible, large scale network routing simulator in support of both academic and business functions.

Section II introduces Internet Background Radiation. This is followed by Section III details the approach to meeting the requirement for injecting packets into the simulated network; the decision to make use of input in the form of a packet capture is discussed, with results of testing of the developed system occurring in Section IV. The paper concludes with Section V.

II. Background

Internet Background Radiation (IBR) [3] is essentially network traffic which is fundamentally non-productive in nature; that is, it can only be characterised as network traffic which has reached a destination not purposed to receive it. IBR is a common occurrence across networks which form part of the Internet's address space, and is the result of either: misconfiguration; or network traffic which has been generated with malicious intent. Whilst misconfigurations account for some IBR, overall its contribution is only marginal .

Misconfiguration encompasses erroneous entries in the configuration of destination addresses for services, applications, routing, and so on; as well as errors which might arise at a hardware level — such as the memory corruption of one or more bits — which may be attributed to environmental factors such as heat.

A far larger contributor to IBR is that of malicious traffic; scanning, worms, and backscatter meet this criterion [3]. Scanning is the act sending network probes to an address, or range of addresses, in order to solicit host information; a subset of which is operating system fingerprinting. This information may be of value in carrying out an attack; if it is revealed that a host, or a service running on a host, is susceptible to a known exploit, an attacker could then proceed to follow up with a malicious payload [4]. The act of scanning is often highly automated and while it is always a constant factor in IBR, it is seen more frequently when new exploits, which have yet to be patched, emerge.

Worms, to some degree, exhibit much the same behaviour as scans do; but only insofar as generating scanning traffic. Worms are self-propagating, and once having infected a host, will scan for other hosts which suffer from the same vulnerability. Depending on how wide-spread the vulnerability being used for infection is, worms can infect a network at an exponential rate until such a time as the vulnerability is patched [5]. The total contribution a worm might make to IBR will be sizably more so, at the time of an outbreak.

Whereas denial of service (DOS), and distributed denial of service (DDOS) attacks are not directly classified as IBR, they indirectly make a sizable contribution to its composition. This is attributed to these attacks originating from spoofed Internet addresses. This traffic has been termed as backscatter, and it refers to not the traffic generated by the attack itself, but rather to the traffic which is generated by the victim, in response to an attack [6]. Imagine the effect of one or more hosts spamming another with TCP SYN packets; an attack which has been termed a SYN-flood. Imagine still, that these packets had been manipulated so as to hide the true identities of the hosts involved in performing the attack; this would entail modifying the relevant IP and TCP source headers so as to reflect an address not belonging to the attacker. The victim would in turn respond to each SYN packet with a SYN-ACK, in an attempt to complete the three-way handshaking procedure previously described. This SYN-ACK traffic would then be routed across the Internet, directed not to the attacker, but to the address which has been falsified.

Historically, IBR has been seen to consist primarily of TCP traffic; of which the vast majority comprised of TCP SYN packets; followed by TCP RST packets. ICMP traffic was seen to make a notable impact only at times when a worm outbreak was on-going; this traffic could be attributed to ICMP echo requests which are used in scans. Similarly UDP traffic was seen to spike at times when there was a worm outbreak with an affinity towards this protocol [3]. In more recent years, this makeup has remained similar, though the amount to which IBR accounts for the total composition of network traffic has since increased dramatically; in fact twice as much as productive traffic in the 6 years which separate the two studies. The emergence and growth of botnets, new services, and the growth of the Internet are all contributing factors [7].

The variation in the composition of IBR originally discussed by [3], and corroborated by [7] are indicative of its highly volatile nature; its makeup is sensitive to the state of the Internet, and as a result, described by the seemingly random, complex, and often malicious, set of events which occur within it over time. The ubiquity of IBR adds an additional level of complexity in analysing Internet traffic, as it can act as a facade to otherwise interesting traffic. Whilst filtering can to some extent lesson the presence of IBR on a network, its variety in composition makes this a difficult task to accomplish [8].

III. Packet Injection

The ability to transmit targeted traffic within a simulation allows for creating a more realistic environment from which many applications could benefit. As an example, should this traffic be such that it can be likened to IBR, locating interesting traffic becomes a task that is no longer trivial; and as a result, an exercise which merits some skill. Moreover, the addition of the feature advocates the modules' usage for tasks which may be more focused on investigating the behavioral characteristics pertaining to a given capture. In this application, the module only facilitates the process through targeted delivery; any investigation into the traffic, or traffic generated in response to the traffic, would need to occur outside of the simulation.

A. Input Format

In an effort to not limit the applications the addition might serve, only a packet capture *.pcap* needs to be provided; the purpose for which, is then inherent in the capture itself. This omits the need for supporting configuration files; as one can tailor a capture to their needs prior to calling upon the modules replay feature.

Due to the wide support of the chosen packet capture format, a myriad of tools exist purposed towards capturing and storing network traffic; perhaps most notably so, Wireshark[1] and Tcpdump[2]. Tcpdump offers far more than just packet capture, is perhaps better suited to those more inclined towards the command-line. Wireshark on other hand provides a graphical interface for both viewing captured and live traffic; and is supported on both Windows and Unix platforms. Software such as *bittwist*[3] and *tcpreplay*[4] allow one to modify the contents of a packet capture, through providing the tools *bittwiste*, and *tcprewrite* respectively.

[1] https://www.wireshark.org/
[2] http://www.tcpdump.org/
[3] http://bittwist.sourceforge.net/
[4] http://tcpreplay.synfin.net/

B. Packet Injection Design

Whilst the *tcpreplay* package provides the ability to modify network traffic, its primary use is in replaying a given packet capture on a specific network interface. When coupled with *netfilter*s ability to intercept network traffic, the two services working in tandem offer an attractive alternative to parsing a packet capture with a *procfs* callback.

However, *tcpreplay* makes use of the Linux *packet*[5] library which permits software operating in user-space to receive, construct and transmit traffic at the device driver level — layer 2 of the network stack. As a result of this, *netfilter* (Which operates at layer 3 of the network stack), is unable to intercept the traffic.

In order to address the issue, whilst at the same time still make use of the method in place, *libdnet*[6] was ultimately used. The Python library provides, among other things, raw IP packet (Layer 3) transmission. This results in the generated traffic being discoverable to *netfilter*.

Subsections III-C and III-D, provide more details pertaining to the feature implementations. These subsections are distinguished from one another as a result of the different logical function each provides.

C. PCAP Parsing and Transmission

In selecting Python for the task of PCAP parsing, and the transmission of the contents contained therein thereafter, the feature is well-positioned for integration with the Python web service which acts as the user interface for the Simulator. A library which greatly simplifies this task is *dpkt*[7].

Parsing PCAPs is managed with the use of **dpkt.pcap.Reader()**, which takes as an argument a PCAP *file*. The method implements an iterator, which returns upon each iteration, a tuple containing the *time stamp*, and *packet buffer* associated with each packet within the PCAP. Thereafter the libraries **dpkt.ethernet.Ethernet()** function can be passed the previously obtained *packet buffer* in order to produce a *frame* object.

Conveniently, in order to obtain the *IP datagram* contained within the *frame*, one needs only to address the objects *data* attribute. Thereafter, the *dnet* library can be used in computing the datagram's checksum, thereafter the returned *buffer* can be transmitted. A code snippet which provides a simple example of the aforementioned process is provided in Listing 1.

By default, a capture will be played back at the same rate the packet capture was recorded. This is managed through differencing the *time stamp* obtained with each packet in the capture; and then calling upon **time.sleep()**. In order to mitigate foreseen problems in the function's responsiveness for even relatively small packet captures, packet replay takes place in a thread. Should another capture be loaded, the current running thread is stopped, and a new thread, containing the new capture, is started.

[5]http://man7.org/linux/man-pages/man7/packet.7.html
[6]https://pypi.python.org/pypi/dnet/1.12
[7]https://pypi.python.org/pypi/dpkt

Listing 1. Code Sample

```
socket = dnet.ip()
pcap = file('sample.pcap', "rb")

for ts, pkt in dpkt.pcap.Reader(pcap):
  frame = dpkt.ethernet.Ethernet(pkt)
  iph = frame.data
  # IP header can be modified by
  # assigning a value to iph."field name"
  buff = dnet.ip_checksum(str(iph))
  socket.send(buff)
```

For the purpose of enhancing the feature, two additional options were added; the first of which is looping capture playback. If this option is set, playback will continue indefinitely, or until another capture is loaded. The second option is fast playback; if this option is set, **time.sleep()** is not called between successive iterations.

D. Netfilter Hook for Traffic Replay

In order to communicate to the Kernel, that the parsed capture traffic is destined for the simulation, each packet has the TTL value within its IP datagram set to *0* prior to being transmitted. Through utilising a *netfilter* hook, all outbound traffic first has the value of its TTL checked, prior to any additional lookups being performed. This novel method mitigates the need to perform a lookup on the source address, of each and every packet being transmitted, which may otherwise prove resource intensive.

If this value is *0*, the packet will subsequently have its source address looked up in the radix tree which stores the simulations *node*s; otherwise the packet continues its traversal up the network stack. Should a matching *node* be found, where the match has similarly been marked as *virtual*, a *pkt* will be created which will have as its current node field, a pointer to the *node* which reflects the packets point of entry into the simulation.

As each virtual node has its own initial TTL value associated with it, the *pkt*s value is updated so as to no longer contain *0*. Much like inbound packet interception, the created *pkt* has its other relevant information set prior to being scheduled for routing. Details pertaining to the routing process are covered in detail in [2] and are not addressed in this work.

IV. Packet Injection Applications and Testing

Packet injection was introduced in Section III, in which its application in facilitating realistic training simulations was discussed. Another application, in which it may be found to be well-suited, is in malware analysis. As a result of the simulator's ability to isolate its traffic from legitimate network traffic — which may be being transported over the same network medium — malware can be executed, and its behaviour analysed dynamically. An example in which this capability may prove particularly useful is in simulating a worm outbreak. A worm's behaviour, targets, and and rate of

infection, can be examined in a manner which both practical and reproducible, whilst at the same time contained.

For the purpose of testing the simulator's ability to inject targeted traffic, a far more benign traffic sample was used. This sample was obtained using Wireshark, and contains nothing other than the results of having pinged www.google.co.za from a network host with the local IP address 10.0.0.60. This information is presented graphically in Figure 1. In order to appropriate this information to the example configuration, each packet had its source and destination addresses modified so as to target a physical host bound to the virtual address 10.0.0.1 from the virtual address 146.231.128.43; an address which in previous examples was representative of www.ru.ac.za.

This was accomplished through using *bittwiste*, a tool mentioned previously in Section III-A. Listing 2 presents the command used in meeting this purpose. The *'-s'* and *'-d'* options, when used in conjunction with the *'-T ip'* argument, permit the source and destination fields present each datagram to be modified. In the case of Listing 2, each occurrence of the IP address 10.0.0.60 will be replaced with 146.231.128.43, and 216.58.223.35 (www.google.co.za) with 10.0.0.1.

The modified packet capture — *sim_ping.pcap* — was then uploaded through the simulators web interface, and Wireshark started on the target itself. The results of which are provided in Figure 2. As can be seen, the capture which corresponds to Figure 1, having been replayed, correctly contains the four ICMP ECHO_REQUESTs with which it was targeted. Each destination address reflects not the virtual address of the target, but the physical address (10.42.0.21) to which it has been bound. This is a result of the simulator having modified each IP header so as to reach its destination over the physical network.

Notably, the four ICMP ECHO_RESPONSEs (observable within within Figure 2 as packet numbers 6, 8, 10, and 12) are a result of the target having responded to the ECHO_-REQUESTs it has been targeted with, and not as a result of these responses themselves having been injected into the simulation. This result is important as it would not make sense, when assessing the response behaviour of a target if this behaviour is not in fact representative of the target itself.

In addition, Figure 2 provides more information from which some general observations can be made. These observations are listed below:

1) Whilst the virtual host 146.231.128.43 was configured to have an initial TTL of 64, there are three gateways which reside on the route. With each gateway traversed by these packets, their TTL has been appropriately decremented. A reported TTL of 61 is therefore correct. This result proves useful due to the manner in which packets are marked when being injected, as detailed in Section III-D.
2) Whilst it is only the first packet that has all of its information presented in detail, each injected packet has had its checksum correctly recomputed. Confirmation is provided by the uniform colouring of each of the packets presented in Figure 2.

TABLE I
RELATIVE TIMING DIFFERENCES

REQUEST	OrigTimestamp	Avg Injected Timestamp	Difference
1	0.000000	0.000000	0.000000
2	0.998204	0.999485	0.001281
3	2.003169	2.005797	0.002628
4	3.003156	3.007157	0.004001

3) Each injected packet has maintained its relevant information. Whilst the sequence number in this case is not particularly useful, it in addition to the type, and code, fields, provides verification that the characteristics held by each packet have been left unmolested.

Another result of import is the timing information associated with the injected traffic. It was a desired feature within this implementation that injected traffic reflect the same delay in transmission as that present in the packet capture being replayed. In order to test whether or not this holds true, this same packet capture was replayed 100 times. This was accomplished using the features *loop* option. Each of the four ICMP ECHO_REQUESTs then had the difference between their timestamp and the first timestamp in the set calculated. Each respective packet then had an average timestamp calculated. This information, as well as the original values of the capture, are presented in Table I for comparison.

As can be seen in Table I, there is little variance between each injected packets average timestamp, and that of the original values. However, as can also be seen, each successive packet produces a *difference* which is slightly larger than it was previously. Whilst these *differences* remain in the region of only a few milliseconds, this deviation would be seen to increase were the capture to be carried out over a longer period of time. This *delay* can be attributed to the processing time required in preparing each captured packet for transmission. This delay is then compounded for each successive packet transmitted by the system.

The amount by which this processing delay skews the captures delivery time appears to be fairly consistent; with each additional packet transmitted, this value is seen to increase by some x. This value can be approximated as the first *difference* obtained in Table I. A better approximation can be made if all of the *differences* are taken into consideration. A simple method for doing this, is by taking all of the *differences*, with the exception of the first, and dividing the result by 6. This allows for creating a weighted average, in which higher *differences* carry more weight; thereby lessening the effects of other factors such as transmission delay. This calculation is provided in Equation 1.

$$\begin{aligned} x &\approx 1 \times \frac{1}{6} \times (1 \times \frac{0.001281}{1} + 2 \times \frac{0.002628}{2} + 3 \times \frac{0.004001}{3}) \\ &\approx \frac{(0.001281 + 0.002628 + 0.004001)}{6} \\ &\approx \quad 0.001318 \quad \text{seconds} \end{aligned} \tag{1}$$

A perhaps more reliable indication of the degree to which this delay affects packet delivery time would be in computing

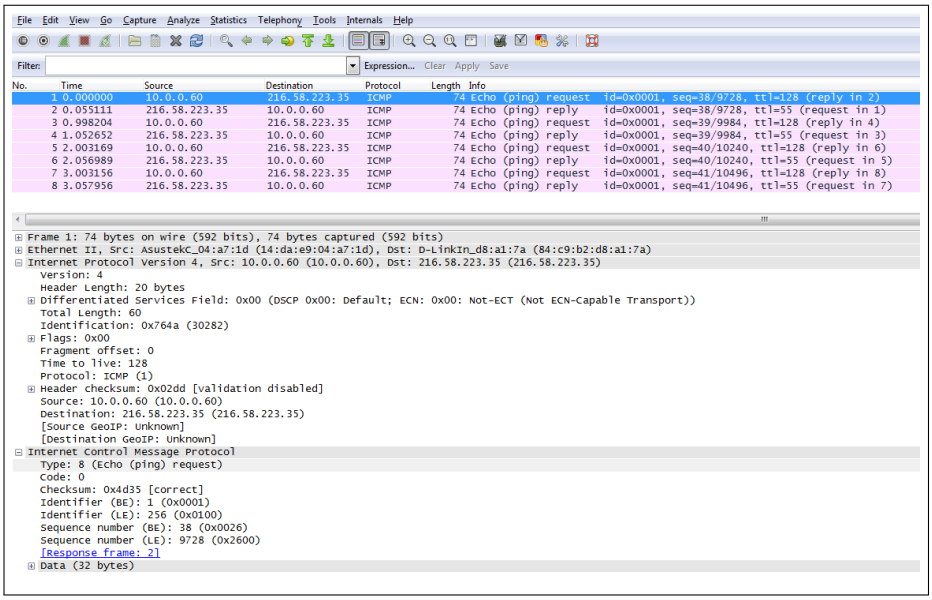

Fig. 1. Wireshark ping capture

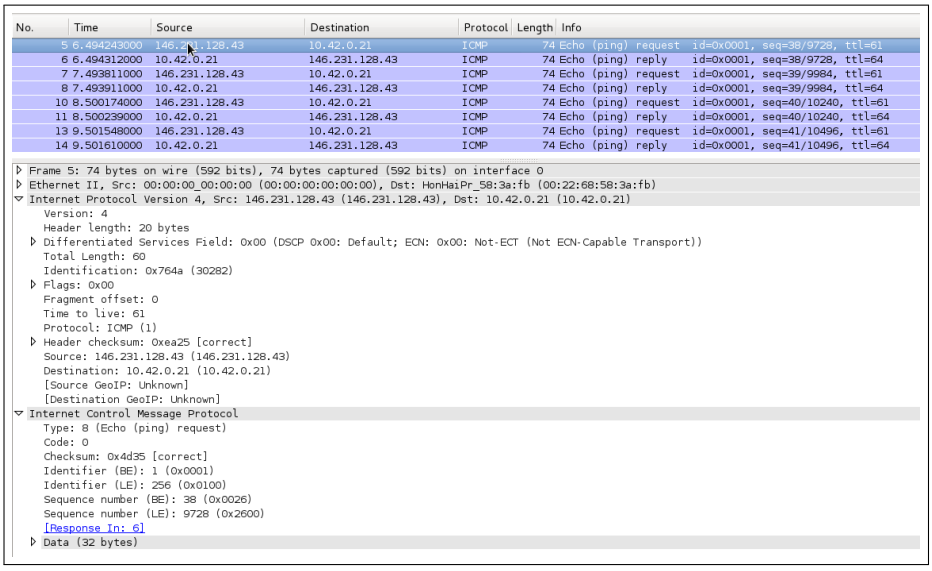

Fig. 2. Wireshark ping capture replay

the number packets, n, it would take to delay transmission by t seconds. In the case of the above, n is representative of the number ICMP ECHO_REQUESTs for which their exists the relationship $t \approx n \times 0.001318$. This relationship is depicted in Figure 3 in which t is plotted against $n \times x$. For the purpose of providing a more definitive approximation, Equation 2 reveals that 45524 ICMP ECHO_REQUESTs would need to be transmitted to delay transmission by one minute. This point has been marked on Figure 3.

$$\begin{aligned} n \times x &\approx 60 \quad \textit{seconds} \\ n &\approx \frac{60}{0.001318} \\ &\approx 45523.520 \\ &\approx 45524 \quad \textit{packets} \end{aligned} \tag{2}$$

Section IV tested the systems packet injection feature. A scenario was presented in which sample capture was obtained, replayed, and recorded. This process was presented in detail so as to allow for the possibility of its recreation. This information was then used to assess the degree to which the feature performed accurately. It was found that each payload had correctly been modified, thereby providing a mechanism for targeted behavioral analysis, as was intended. However, it was also established that a processing delay had been introduced. Whilst this delay was admissible for small packet captures, its nature being compounded lead to increasingly larger values with each additional packet transmitted. This relationship was approximated and presented in Figure 3.

Listing 2. Packet Capture Rewrite

```
~# bittwiste -I ping.pcap \
>       -O sim_ping.pcap \
>       -T ip \
>       -s 10.0.0.60,146.231.128.43 \
>       -d 216.58.223.35,10.0.0.1

input file: ping.pcap
output file: sim_ping.pcap

2 packets (196 bytes) written
```

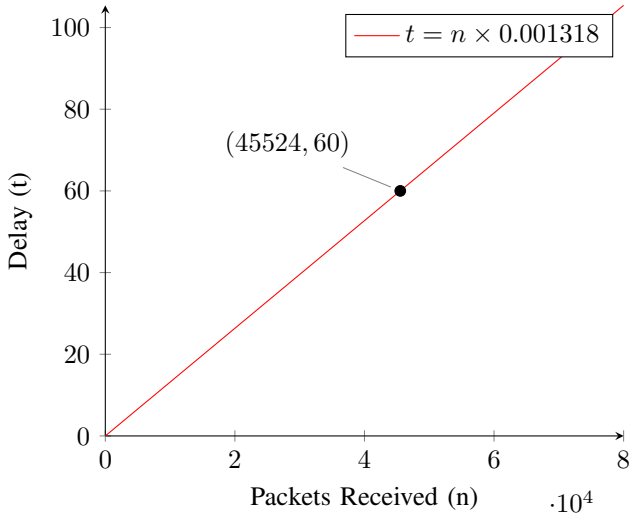

Fig. 3. Processing Delay Skew

V. Conclusion

The systems ability to inject targeted traffic was verified in Section IV where it demonstrated a high degree of veracity where content was concerned. Though timing accuracy was somewhat skewed, the amount by which this was seen to be true was fairly consistent, and thus calculable. Therefore, for applications more demanding in this regard, this delay can be taken into consideration.

A. Future Work

Currently the NKM system only supports IPv4 datagrams, and the TCP, UDP and ICMP protocols. This is expected to be extended to support IPv6 in the near future. The packet injection module described herein can also be extended to provide this support, and extend its own protocol support. Additional work can also be explored in allowing the injection of specific Layer 2 frames for direct delivery to client systems.

Acknowledgement

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

References

[1] Internet World Stats. (2014) Usage and Population Statistics. Miniwatts Marketing Group. http://www.internetworldstats.com/stats.htm. Accessed 11 May 2015. [Online]. Available: http://www.internetworldstats.com/stats.htm

[2] A. Herbert, "Towards large scale software based network routing simulation," Masters Thesis. Rhodes University, South Africa, 2014.

[3] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson, "Characteristics of internet background radiation," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004, pp. 27–40.

[4] C. T. Wai, "Conducting a penetration test on an organization," Tech. Rep., 2002.

[5] C. C. Zou, L. Gao, W. Gong, and D. Towsley, "Monitoring and early warning for internet worms," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 190–199.

[6] K. E. Giles, D. J. Marchette, and C. E. Priebe, "On the spectral analysis of backscatter data." Citeseer, 2004.

[7] E. Wustrow, M. Karir, M. Bailey, F. Jahanian, and G. Huston, "Internet background radiation revisited," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 62–74.

[8] P. Barford, R. Nowak, R. Willett, and V. Yegneswaran, "Toward a model for source addresses of internet background radiation," in *Proc. of the Passive and Active Measurement Conference*, 2006.

Craig Koorn received his Compter Science Honours in 2015 at Rhodes University under Barry Irwin. He is currently working in the Information Security Sector as a penetration tester.