Scaling IPv6 Address Bindings in Support of a Moving Target Defense

Christopher Morrell, J. Scot Ransbottom, Randy Marchany, Joseph G. Tront
Bradley Department of Electrical and Computer Engineering, Virginia Tech Blacksburg, Virginia, USA
Email: {morrell, jransbot, marchany, jgtront}@vt.edu

***Abstract*—Moving target defense is an area of network security research in which machines are moved logically around a network in order to avoid detection. This is done by leveraging the immense size of the IPv6 address space and the statistical improbability of two machines selecting the same IPv6 address. This defensive technique forces a malicious actor to focus on the reconnaissance phase of their attack rather than focusing only on finding holes in a machine's static defenses. We have a current implementation of an IPv6 moving target defense entitled MT6D, which works well although is limited to functioning in a peer to peer scenario. As we push our research forward into client server networks, we must discover what the limits are in reference to the client server ratio. In our current implementation of a simple UDP echo server that binds large numbers of IPv6 addresses to the ethernet interface, we discover limits in both the number of addresses that we can successfully bind to an interface and the speed at which UDP requests can be successfully handled across a large number of bound interfaces.**

***Keywords*—Moving Target Defense; IPv6; Sockets; Networking.**

I. Introduction

Improving security by hiding targets within a network is an active area of network security research. This effort occurs in conjunction with the current use of traditional security methodologies where network and network resources are treated like castles with layered defenses built up around them. A common method to achieve this machine hiding involves leveraging the immense space available within IPv6 in order to move a machine logically around a subnet by reassigning layer 3 addresses at a frequent interval. By coordinating this change in either a scheduled manner, or permitting hosts to calculate addresses for both themselves and their peer, we can ensure that two communicating machines can always find each other in the network. This methodology is often referred to as a moving target defense. There is a fairly small community of researchers working on this problem, including Zhuang et al in [1] and Yackoski et al in [2]. While much of the work that has been done to this point focus on the theories around a moving target defense, researchers at Virginia Tech have designed, implemented, and published a moving target defense algorithm entitled Moving Target IPv6 Defense or MT6D [3].

In its original design and implementation, MT6D was focused solely on peer to peer networks, while generally leaving the much more common client server networks untreated. Our intent in this paper is to take the first steps toward pushing MT6D in the direction of providing client server network support. In an ideal situation, each client would have a distinct address on the server that it would communicate with. Alternatively, a server could maintain a single set of hopping addresses, and every client would use the same address. The former provides better privacy with a higher cost of resources, while the latter is simpler but reduces the privacy provided by the algorithm.

This paper focuses on the first step of identifying and quantifying the scaling problems that will occur when we bind a large number of addresses to a host and permit a large number of clients to communicate with the server on those addresses. Ultimately, we must understand how many clients we can actually support with a single MT6D server. To this end, we must determine how many IPv6 addresses can be supported in the modern Linux kernel and how quickly a server can respond to client requests when it is bound to some large number of addresses.

We explore those two limits by implementing a simple UDP echo server that binds some number of addresses and listens for requests on all of them. Upon receipt of the request, the server should respond with some data. In this paper, we will give a brief treatment of IPv6 and MT6D so that the reader has the requisite understanding of both protocols to further understand this research area. We will follow by explaining our motivation for conducting experiments in the manner that we did. We will discuss the limits that we reached on binding IPv6 addresses in our implementation, followed by a discussion of the ideal parameters for our echo server. Finally, we will discuss our next research steps in the future work section before we conclude.

II. Background

A. IPv6 Overview

Internet Protocol version 6 (IPv6) is the 128-bit replacement to the aging 32-bit IPv4 standard. IPv6 was originally published as part of RFC 2460 [4] in December of 1998, when it was realized that IPv4 would not be able to support the number of IP addresses that would be required as the Internet continued to grow. The biggest benefit provided by the move to IPv6 is the immense growth in the number of addresses available as we move from 32 bits to 128 bits. As a simple demonstration of the vast quantity of addresses in IPv6, there are sufficient addresses that one could assign more

978-1-908320-39/1/$25.00©2014 IEEE

that 667,000 addresses per square nanometer on the surface of Earth, including the oceans. To truly bring this number to scale, realize that the area of the tip of an average human hair is approximately 2 billion square nanometers. For direct comparison, we should see how this compares to the IPv4 addressing scheme that uses only 32 bits. Instead of addressing nanometers as with IPv6, we will address kilometers of the surface of the Earth. In IPv4, each square kilometer on the surface of the planet gets only 8 addresses. We use this extreme difference in size to try to communicate an understanding of the immense space gained by simply adding 96 bits to our IP addresses.

In IPv6, the address is generally split into three pieces, representing different parts or layers of the network. The first 48 bits are used to represent the routing prefix, the next 16 bits represent the subnet identification, and the last 64 bits, also known as the interface identifier (IID), represent the host. This means that on any subnet, there are approximately 2^{64} or $1.845x10^{19}$ distinct IIDs available. In stark contrast, there are only a total of 2^{32} or 4 billion addresses available in the entirety of the IPv4 address space. Due to the limited address space in IPv4, assigning addresses requires a great deal of planning and close control from network administrators in order to ensure efficient usage. In fact, the IPv4 addressed internet relies in large part to RFC 1918 [5] private addresses in order to function. These private addresses permit a large number of addresses to masquerade as a single address. In contrast, it is completely feasible to permit a client to select their own IID in IPv6, since the probability of a collision is almost zero. In fact, according to the birthday paradox, it would take approximately $2^{64/2}$ or just over 4 billion attempts to have a greater than 50% likelihood of a collision inside of a 64 bit IPv6 subnet. This fact permits the majority of IPv6 networks to rely on IPv6 Stateless Address Autoconfiguration (SLAAC) in order to assign addresses to their clients.

SLAAC is an automatic addressing scheme that was formalized in RFC 2462 [6] and updated in RFC 4862 [7] that provides a machine with the ability to generate its own an IPv6 address. In SLAAC, clients ask their local router for the routing prefix and subnet id portions of the address by sending a router solicitation message to the nearest router. The router responds with a router advertisement message which gives the client the first half of their IPv6 address. The host then generates its own IID and appends it to the prefix and subnet id that were received from the router, and assigns the address to its network interface. IID generation algorithms vary, but are usually based on the network interface card's Media Access Control (MAC) address. Since a machine will likely use the same method each time it generates an IID, it is probable that the IID will be the same for a given machine.

B. Moving Target IPv6 Defense

Moving Target IPv6 Defense (MT6D) was introduced as a specific solution to the moving target defense problem, and was originally proposed by Dunlop, et al. in a 2011 paper entitled *MT6D: A Moving Target IPv6 Defense* [3]. MT6D leverages the enormity of the IPv6 address space by allowing multiple hosts to assign themselves new IPv6 addresses at arbitrary but pre-determined time intervals, while they maintain the ability to communicate with each other. As with SLAAC, the large number of addresses available within the IPv6 address space gives an almost zero probability of an address collision occurring.

The addresses used for reassignment are created through the MT6D algorithm, which uses the machine's SLAAC generated or statically assigned IPv6 Interface Identifier (IID), a passphrase, and the current time. Configuration is required so that the machines have a pre-shared copy of the seed IID and the passphrase. The algorithm also requires a relatively close time synchronization, usually within several seconds. Machines are also provided with the seed IID, IPv6 prefix, and subnet ID of the host that they will be communicating with via an MT6D connection. With all of the address generation information available to both hosts, machines are able to calculate not only their own MT6D addresses, but also the MT6D addresses of their peer. By defaults, hosts generate and assign new addresses every 10 seconds. Additionally, each host keeps its previous, current, and next address assigned to its interface in order to prevent packet loss due to latency in the network. MT6D relies on UDP for transport, and simply encapsulates an entire IPv6 packet inside of an MT6D generated UDP datagram and MT6D generated IPv6 packet. The authors have engineered MT6D in such a way that they can keep TCP sessions active, even during MT6D address rotations.

This ability for a machine to logically move throughout the network at will forces a shift in the way an attacker does business. Rather than following the traditional phases of reconnaissance and exploit, MT6D forces an adversary to commit all of their resources to the reconnaissance phase. Additionally, MT6D does not preclude someone from applying standard security defenses to a machine, rather adds a layer of anonymity on the network on top of the standard security.

MT6D was originally designed, and has only been implemented up to this point to function as a peer to peer service. While it is theoretically possible to stretch MT6D to support several more machines, it was never designed with a client server network in mind. While peer to peer networks are an extremely powerful and useful section of the Internet, a great deal of data is exchanged between clients and servers. For example, each time you browse a webpage, upload a file to cloud based storage, or send an email, your machine is a client, interacting directly with a server. It is with this in mind, that we are working towards extending MT6D to support client server networks.

C. Methodology

The first step in designing and building an MT6D server, is to understand what is possible given the capabilities provided in the modern Linux kernel using standard commodity hardware. We must first determine the number of addresses that a machine can bind to its network interface. This will help us

to determine how many clients we can support with a single server. Additionally, we must determine how quickly we can send data to the server considering the number of addresses that it has currently bound. This will help us to determine the maximum possible rate of data as the number of active clients changes.

A number of different methods can be used to bind and unbind addresses within Linux. The simplest is to call *ifconfig* [8] from within your server, and allow the utility to handle the address binding and unbinding for you. However, this is extremely inefficient. Each time *ifconfig* is called, a new process must be spun up in order to handle the request. Alternatively, the server can send *Input Output Control (ioctl)* [9] or *netlink* [10] messages to the kernel. For our experiments, we chose to use *netlink* messages due to their ease of use and speed.

As there are a number of methods for binding and unbinding addresses, there are also a number of methods that can be used to listen for, receive, process, and respond to client requests on a server. One of the easiest and most common methods is to use standard BSD sockets [11]. Sockets are well supported, but require a reliance on the Linux kernel to process packets from the hardware up to the layer at which the socket is listening before being handed over to the server. Rather than using sockets, it is possible to use zero copy networking [12], where a server is permitted to reach past the kernel all the way to the device driver, reducing the reliance on the kernel's processing, reducing the need for multiple context switches, and reducing the number of copies that are required when a packet is moved from the hardware to the server. While zero copy networking is more efficient than standard sockets, we decided to use sockets for these experiments, leaving zero copy networking for future work.

D. Network Service Performance Metrics

A number of network services may find some benefit in being obscured with a moving target defense connection. If I am surreptitiously attempting to share data with someone, it would be beneficial to have my location on the network obscured. I could even use Voice over IP (VoIP) services and wish to have the location of my phone obscured within the network. While both of these examples could potentially benefit from implementation of a moving target defense, the metrics used to define the usability of each are significantly different. If I am uploading a file or downloading a webpage, some delay in the connection with my server may be acceptable, while delay in connection when dealing with a real time service such as VoIP must be much less.

In his book *Usability Engineering* [13], Jakob Nielsen discusses acceptable response times in order to give a user different feelings about the content that they are interacting with. According to Nielsen, responses received within a 0.1 second window make the user feel as though the system reacts instantaneously. The author states that a 1 second response time is the generally accepted upper bound for the system to not interfere with a user's flow of thought, although they will notice that there is some delay in the system. We take these ideas and rough numbers as inspiration and desire to keep our server response time as close to the 0.1 second mark as possible. In future sections, we will see how this will help us to dictate how many addresses and thus how many clients we are able to support on our server.

In the next two sections, we will discuss in more depth the implementation of our server and the results of the experiments that were run in support of our understanding the limits of building a server that listens on numerous IPv6 addresses at the same time.

III. Time to Bind Addresses

How many IPv6 addresses can be bound to a single machine at any give time, while still permitting the machine to function on the network? Identifying this limit will give us an upper bound of the number of clients that our MT6D server will realistically be able to support at a single time. In order to discover this limit, we built a simple program in C that generates a list of IPv6 addresses, binds those addresses to the machine's eth0 interface, and once complete, unbinds those addresses from the eth0 interface.

We used a Dell OptiPlex 9020 running 64-bit Debian Wheezy [14]. The machine has an Intel i7-4770 processor running at 3.4Ghz, 16GB of RAM and an Intel I217-LM Gigabit Ethernet network interface card. The experiment consists of a program written in C that uses the rtnetlink [10] library in order to send messages the the kernel for IP Address binding and unbinding. The program begins by generating a list of *n* IPv6 addresses, where *n* varies from 1 to 70000 in increments of 5000. We continue sending netlink *RTM_NEWADDR* (bind) messages for each address as quickly as possible. Once all addresses were successfully bound, we send *RTM_DELADDR* (unbind) messages as quickly as possible. Address generation, binding, and unbinding were timed individually, and each measurement was averaged across the total number of addresses that were bound and unbound. Each set of parameters was run 5 times and averaged, resulting in the data that is displayed in Fig. 1.

We found that the average address generation time increased as addresses increased as expected, and our implementation had a maximum time of 2 milliseconds to generate 70000 IPv6 addresses. Additionally, we found that the average time to generate an address did not increase, and was between 16 and 48 nanoseconds regardless of the number of addresses that we generated.

We found that there was a roughly linear increase in time as we increased the number of addresses to bind up to the 55000 address mark. Above 55000 addresses, we found an exponential increase in time for each address bound. The average time to bind addresses increases from 176 microseconds at 1 address to 3.9 milliseconds at 50000 addresses. After 55000 addresses, we quickly increase to 11 ms, 50 ms, and 101 ms, for 60000, 65000, and 70000 addresses respectively. The total time required to bind addresses shows a dramatic increase

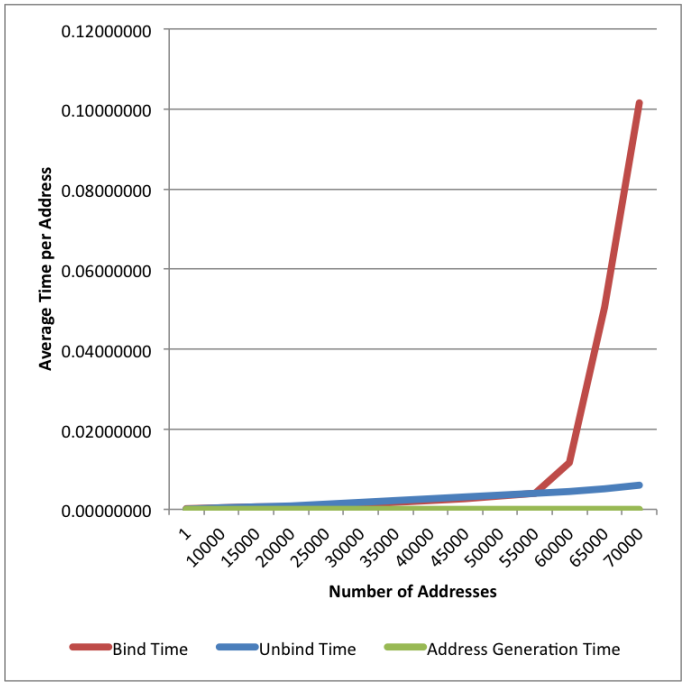

Figure 1. This chart shows the average time required to generate, bind, and unbind an IPv6 address based on the given number of addresses.

from 216 seconds for 55000 addresses to 7108 seconds for 70000 addresses.

While time to bind has a dramatic increase above 50000 addresses, time to unbind continues to follow a linear time increase all the way up to 70000 addresses. At its slowest, the average time to unbind an address was approximately 5 milliseconds.

We also ran some basic computer and network functionality tests after addresses were bound in order to discover whether our limitation was the actual binding of the addresses, or the memory required to keep the addresses bound. Even with 70000 addresses bound to our machine, we did not see a noticeable slowdown in the performance of our machine through both basic computing and performing a large file download from the internet.

Experimental results lead us to to the determination that our current implementation has an upper bound of approximately 55000 actively bound IPv6 addresses. It is possible to bind more than 55000 addresses, although the time required per address becomes longer than our desired address rotation period as described in the MT6D section. This upper bound has also helped us to determine our limit when conducting our next set of experiments, which focus on tuning our server for best performance.

IV. Server Response Time

Previous experiments have provided us with a clear understanding of what the limits of our implementation are in regards to the maximum number of addresses that we can manage. We must now determine what the maximum rate of data receipt is for the server. In order to determine these values, we built a UDP echo server that listens on all bound IPv6 addresses on UDP port 3540. When a request is received, the server responds with the requested data which has been calculated at ten times the size of the request. This increase in size of the response is intended to replicate the standard request/response size ratio that is inherent in most web based services. In our implementation, we use standard BSD sockets [11] with *AF_INET6* and *SOCK_DGRAM*.

Once addresses are bound, we spawn a listener thread and a worker thread. The listener thread simply receives UDP requests and queues them in a POSIX message queue [15] as quickly as they are received. The worker thread looks for messages in the queue, and upon receipt of a message, sends the data back to the requesting client ten times. As return packets are built, we ensure that the source address is the actual address where packet was received, rather than allowing the kernel to address the packet for us. We discovered that by default, the kernel that we were using would use the highest IPv6 address as the source if not modified otherwise. Since the worker is sending ten times the data that the listener is receiving, it is apparent that the worker thread is the slowest part of the server. Earlier implementations of the server attempted to leverage multithreading, but we found that contention on the call to BSD sockets *sendmsg()* resulted in much poorer performance.

A production implementation of our server would be concurrently communicating with n clients, where n is some number up to 55000. Rather than dealing with the issues that come with managing a group of clients at that scale, we chose to use some simple traffic generation tools in order to replicate the real world. We began by building a client traffic generator around the Python packet generation tool, Scapy [16], but found that the Python interpreter did not generate traffic quickly enough to put any stress on the server. We attribute this primarily to the limitation of using the Python interpreter, rather than using compiled code. Through further research, we discovered a C++ library that provides the ease of use of Scapy with the efficiency of compiled C++ code entitled libtins [17]. When our client was permitted to send traffic without delay, we were able to achieve a rate of approximately 150,000 packets per second (pps). We used a simple sleep timer within the client to allow us to control our rate of transmission in the range of 1900 to 150000 pps.

In all of our experiments, we had the server bind n addresses and create a UDP socket in order to listen on all addresses before the client was allowed to proceed. Once the server was finished and idly listening, the client would send 250,000 UDP requests to the server at the dictated packets per second interval. Destination IP addresses were selected at random from the pool of addresses that the server had bound. We counted receipt of fewer than 90% of those 250,000 requests to be a failure, and discounted those experiment parameters from the data collection. For this set of experiments, we varied packets per second in the range of 1900 to 150000 pps and the number of addresses in the range from 1 address to 55000 addresses in increments of 5000 addresses. The range of packet

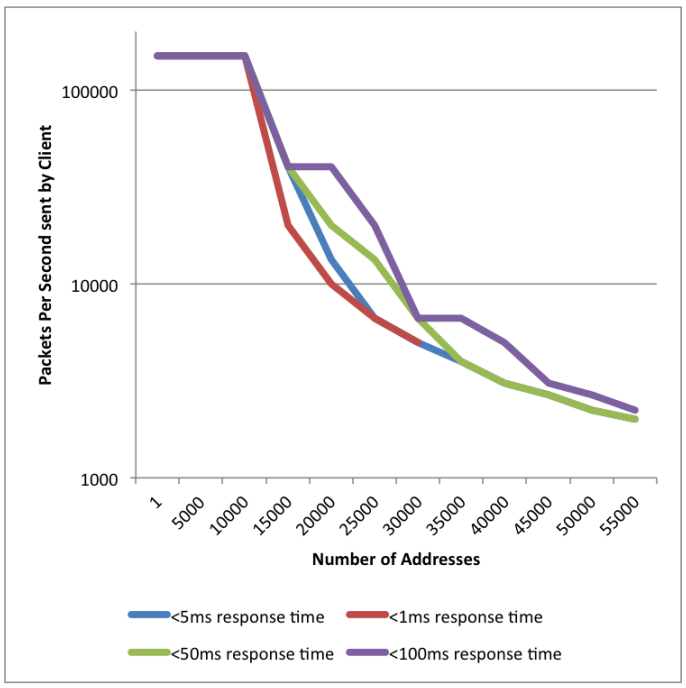

Figure 2. This chart shows the rate of data transmission in packets per second against the number of addresses bound on the server resulting in the ideal values based on the necessary server response time. The Y axis data is represented using a logarithmic scale.

speed does not fall on clean numbers due to the fact that we were varying sleep time on the server, which indirectly impacted our packets per second rather than dictating a specific packet per second number. Experiments were run on each set of parameters five times and results were averaged.

In Fig. 2 we can see the results of the described experiments. On the X axis, we see the range of addresses bound to the ethernet interface from 1 to 55000, while the Y axis gives us the range of packets per second from 1900 on the bottom to 150000 at the top. The data follows a nearly logarithmic function, so the Y axis is presented in a logarithmic scale with 200000 as the maximum value on the axis and 1000 as the minimum. The four lines represent the combination of pps and address number that resulted in a particular average response time. As the legend demonstrates, red represents an average response time of less than 1 ms, blue represents an average response time of less than 5ms, green represents an average response time of less than 50ms and purple represents an average response time of less than 100ms.

We show the results with differing average response times due to the fact that different services require a different response time from a server. For example, real time services have much lower tolerance for delay from the server, while other non-realtime services such as e-mail are more tolerant of delays from the server.

In Fig. 2, we can see that we did not have any tests with more than 30000 addresses that were able to respond in less than 1 millisecond, no matter the rate of requests. Once we consider a 5 millisecond response time as acceptable, we have results that we can analyze all the way to 55000 addresses. Unfortunately, we can see that we have a maximum possible data rate of 2000 packets per second when we are bound to 55000 addresses. This means that on average, our server would only be able to support 55000 clients if their individual data rates averaged one packet every 27 seconds. This is of course fully unacceptable in almost every situation where client server interaction is required.

We can see though that performance increases as the number of addresses decreases, reaching a maximum supportable data rate of 150000 packets per second when 10000 or fewer addresses are bound to the server. We find in counter to the extreme case of 55000 addresses that a situation where 10000 addresses are bound, our clients can sustain an average data rate of approximately 150 packets per second. This of course is still not ideal, but does give us a base line from which to build.

By comparing our results with the desired results described in section II.D., we see that we must limit our server to approximately 10000-15000 addresses bound. While this is a great step forward in building a moving target defense client server network, it is still a severe limitation that we would like to remove. In the next section, we will discuss in some detail the next steps we plan to take in order to remove or reduce these limitations.

V. Future Work

Two primary factors limit the performance of our server. These factors are the time required to bind an address and the time required to receive, process, and reply to a UDP request. We realize that we may not be using the most efficient techniques in order to execute these tasks, so we plan to continue research in these areas to discover, implement, and test alternative methods that may improve the performance of our server.

As discussed in section II.C., there are several methods that can be used to bind IP addresses to an interface. We only analyzed two, focusing on either using *ifconfig* or sending *netlink* messages. As reported, we found that there was a great performance increase by moving from the former to the latter, but we plan to explore *ioctl* in more depth. We will conduct a direct comparison between using *netlink* messages and *ioctl* in order to more clearly determine which is the preferred method in regards to performance. Additionally, there is the possibility that we could bypass the utilities that are provided by the operating system and reach directly into the data structures that hold IP addresses and manipulate them manually. Of course, there is the potential for great risk in using this method, and more research is required to determine if it is feasible. We must determine what the risks are and if those risks provide us with some performance improvement over the methods that we are currently using.

Also in section II.C., we used standard BSD sockets with *SOCK DGRAM* UDP sockets to send and receive data between the machines on our network. We know that this is not the most efficient means of passing data onto a network, but

it is the easiest and most well supported. We plan to identify and explore several other methods that can be used to send and receive data and compare them with UDP sockets to identify the method that gives us the best performance possible.

In particular, we will explore the use of raw sockets[11], zero copy networking[12], [18], and the PF_Ring framework [19] in order to improve server efficiency. Raw sockets will allow us to build our network datagram manually, rather than relying on the Linux kernel. It is possible that we could build our IPv6 and Ethernet headers more efficiently than the kernel does, thus resulting in faster data processing. Zero copy networking is a method in which the ring buffer that is normally used by the kernel's network stack for send and receive queues is moved from kernel space into user space. This simple technique eliminates the need for the server to copy data from kernel space to user space on receipt or to copy data from user space to kernel space on send. This technique can also be implemented in such a way that system calls are nearly eliminated, thus resulting in far fewer time consuming context switches. The final method that we plan to explore is PF Ring, which is really a framework that exploits the power zero copy networking. PF Ring implements most of the low level code required to make zero copy networking function, thus abstracting much of the problem away from the server implementer.

VI. Conclusion

We have shown the limitations in building a server that must bind to and listen on many IPv6 addresses. This particular problem is something that is exceedingly unique in the world of IPv6 and moving target defense. We believe that we have shown that building an IPv6 moving target defense server is possible, although there are some severe limitations as the number of clients and expected bandwidth grows.

In section III, we focused on binding addresses to our server, and the time required to execute that task as the number of addresses grew. Based on our current implementation, we found a hard limit of 60000 addresses, thus giving us a very clear maximum number of clients that we will be able to support with individual addresses on the server for each client. As discussed in the future work section, we plan to explore alternative methods to bind these addresses to the interface in order to improve the efficiency of our implementation with the ultimate goal of raising that hard limit.

Additionally, in section IV, we discussed the response time of the server given a fixed number of bound addresses. In our current implementation, it would appear that we would be best served by limiting the number of clients that we support to under 15000-20000. Should we set that as a limitation, we have the freedom to ignore bandwidth limits. If we are willing to implement bandwidth limits to our clients or if we are dealing with a network service that requires only occasional data transmissions, we may be able to increase the number of clients that we support. We plan to push this work forward through exploring different methods to pass data from the NIC to the server and back to the NIC again with the ultimate goal of increasing efficiency to the point that bandwidth limits are not a concern until we get to a significantly larger number of clients.

References

[1] R. Zhuang, S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal, "Investigating the application of moving target defenses to network security," in *2013 6th International Symposium on Resilient Control Systems (ISRCS)*, 2013, pp. 162–169.

[2] J. Yackoski, J. Li, S. A. DeLoach, and X. Ou, "Mission-oriented moving target defense based on cryptographically strong network dynamics," in *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, ser. CSIIRW '13. New York, NY, USA: ACM, 2013, pp. 57:1–57:4. [Online]. Available: http://doi.acm.org/10.1145/2459976.2460040

[3] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront, "MT6D: a moving target IPv6 defense," in *MILITARY COMMUNICATIONS CONFERENCE, 2011-MILCOM 2011*, 2011, pp. 1321–1326. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6127486

[4] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460 (Draft Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946. [Online]. Available: http://www.ietf.org/rfc/rfc2460.txt

[5] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear, "Address Allocation for Private Internets," RFC 1918 (Best Current Practice), Internet Engineering Task Force, Feb. 1996, updated by RFC 6761. [Online]. Available: http://www.ietf.org/rfc/rfc1918.txt

[6] S. Thomson and T. Narten, "IPv6 Stateless Address Autoconfiguration," RFC 2462 (Draft Standard), Internet Engineering Task Force, Dec. 1998, obsoleted by RFC 4862. [Online]. Available: http://www.ietf.org/rfc/rfc2462.txt

[7] S. Thomson, T. Narten, and T. Jinmei, "IPv6 Stateless Address Autoconfiguration," RFC 4862 (Draft Standard), Internet Engineering Task Force, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4862.txt

[8] ifconfig - linux man page. [Online]. Available: http://man7.org/linux/man-pages/man8/ifconfig.8.html Access Date: 15 Sept 2014.

[9] ioctl - linux man page. [Online]. Available: http://man7.org/linux/man-pages/man2/ioctl.2.html Access Date: 15 Sept 2014.

[10] netlink - linux man page. [Online]. Available: http://man7.org/linux/man-pages/man7/netlink.7.html Access Date: 15 Sept 2014.

[11] socket - linux man page. [Online]. Available: http://man7.org/linux/man-pages/man2/socket.2.html Access Date: 15 Sept 2014.

[12] J. Chase, A. Gallatin, and K. Yocum, "End system optimizations for high-speed tcp," *Communications Magazine, IEEE*, vol. 39, no. 4, pp. 68–74, Apr 2001.

[13] J. Nielsen, *Usability engineering*. Elsevier, 1994.

[14] Debian wheezy homepage. [Online]. Available: https://www.debian.org/ports/amd64/ Access Date: 15 Sept 2014.

[15] mq_overview - linux man page. [Online]. Available: http://man7.org/linux/man-pages/man7/mq_overview.7.html Access Date: 15 Sept 2014.

[16] Scapy. [Online]. Available: http://www.secdev.org/projects/scapy/ Access Date: 15 Sept 2014.

[17] Libtins packet crafting and sniffing library. [Online]. Available: http://libtins.github.io/ Access Date: 15 Sept 2014.

[18] J. Song and J. Alves-Foss, "Performance review of zero copy techniques," *International Journal of Computer Science and Security (IJCSS)*, vol. 6, no. 4, p. 256, 2012.

[19] Pf_ring. [Online]. Available: http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/ Access Date: 15 Sept 2014.