Optimizing a Network Layer Moving Target Defense by Translating Software from Python to C

Owen Russell Hardman

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

In

Computer Engineering

Joseph G. Tront, Chair

Randolph C. Marchany

A. Lynn Abbott

December 4, 2015

Blacksburg, VA

Keywords: IPv6, Moving Target Defense, Optimizing, Python, C

Optimizing a Network Layer Moving Target Defense by Translating Software from Python to C

Owen R. Hardman

ABSTRACT

The security of powerful systems and large networks is often addressed through complex defenses. While these types of defenses offer increased security, they are resource intensive and therefore impractical to implement on many new classes of networked systems, such as mobile phones and small, embedded network infrastructure devices. To provide security for these systems, new defenses must be created that provide highly efficient security. The Moving Target IPv6 Defense (MT6D) is a network layer moving target defense that dynamically changes Internet Protocol version 6 (IPv6) addresses mid-session while still maintaining continuous communication. MT6D was originally written in Python language, but this implementation suffers from severe performance limitations. By translating MT6D from Python to C and taking advantage of operating system specific application programming interfaces (APIs) and optimizations, MT6D can become a viable defense for resource constrained systems.

The Python version of MT6D is analyzed initially to determine what functions might be performance bottlenecks that could be performed more efficiently using C. Based on this analysis, specific parts of the Python version are identified for improvement in the C version by either using functionality of the Linux kernel and network stack or by reworking the code in a more efficient way. After this analysis, the information gathered about the Python version is used to write the C version, using methods specific to a moving target defense to capture, analyze, modify, and tunnel packets. Finally, tests are designed and run to compare the performance of the Python and C versions.

Dedication

This work is dedicated to my uncle, C. Stanley Hardman, who passed away in 2014 after battling cancer. Stan was a wonderful man that I respected for his business prowess, his calm demeanor under pressure, and his fairness. I always appreciated his willingness to listen to me, and to mentor and encourage me. I cherish the memories that I have of him, and I miss him.

Acknowledgements

I would like to thank all of the people who helped me throughout my graduate experience. My advisor, Dr. Joseph Tront, has been patient, friendly, and supportive. I would not have been able to complete this work without his help. Randy Marchany was also helpful and encouraging and allowed me to make use of the equipment and resources in the IT Security Office. I appreciate Dr. Lynn Abbott for taking the time to serve on my committee. I would also like to thank Matthew Dunlop, Stephen Groat, and others in the IT Security Office for their collaboration and help. Mary Brewer also helped to keep me on track.

Finally, I would like to thank my parents, Fred and Martha, for their continuous love, support, and encouragement and their willingness to help whenever needed.

Table of Contents

Table of Figures

Table of Equations

Table of Tables

List of Abbreviations

AES ..... Advanced Encryption Standard

ASIC ..... Application Specific Integrated Circuit

API ..... Application Programming Interface

CBC ..... Cipher-Block Chaining

CPU ..... Central Processing Unit

DDR ..... Double Data Rate

DHCP ..... Dynamic Host Configuration Protocol

DHCPv6 ..... Dynamic Host Configuration Protocol for IPv6

DUID ..... DHCP Unique Identifier

FPGA ..... Field Programmable Gate Array

GIL ..... Global Interpreter Lock

HTTP ..... HyperText Transfer Protocol

IANA ..... Internet Assigned Numbers Authority

ICMP ..... Internet Control Message Protocol

ICMPv6 ..... Internet Control Message Protocol for IPv6

I/O ..... Input and Output

IP ..... Internet Protocol

IPsec ..... Internet Protocol security

IPv4 ..... Internet Protocol version 4

IPv6 ..... Internet Protocol version 6

ITU ..... International Telecommunication Union

MAC ..... Media Access Control

MTD ..... Moving Target Defense

MT6D ..... Moving Target IPv6 Defense

MTU ..... Maximum Transmission Unit

NDP ..... Neighbor Discovery Protocol

NTP ..... Network Time Protocol

RAM ..... Random Access Memory

SHA ..... Secure Hash Algorithm

SLAAC ..... StateLess Address AutoConfiguration

TCP ..... Transmission Control Protocol

UDP ..... User Datagram Protocol

USB ..... Universal Serial Bus

VoIP ..... Voice over IP

Chapter 1: Introduction

As more devices connect to the Internet, the threat of attack, information theft, and loss of privacy to network connected systems increases. Several recent events, such as the theft of credit card information from Target [1] and revelations of spying by the National Security Agency [2] in 2014, add urgency to these problems of security. Typically, the security of powerful systems and large networks is addressed through complex, resource intensive defenses. However, many new classes of networked systems, such as mobile phones and small, embedded network infrastructure devices, have slower processors, less memory, and power usage constraints, which make resource hungry defenses impractical to implement. To address the security and privacy of these network devices, new system defenses must be created that provide highly efficient security.

A new class of complex system protection is the moving target defense (MTD). A MTD involves a controlled, but seemingly unpredictable, change to the attack surface of a system that increases the uncertainty and perceived complexity for an attacker. These near constant changes reduce the window of opportunity for an attack and increase the cost and time required to probe and develop an attack. With a network layer MTD in Internet Protocol version 6 (IPv6), the dynamic parameter providing entropy to the MTD is the address, utilizing the extremely large address space in IPv6. While the increased entropy of a MTD provides security benefits, it also requires significant system resources to compute the mutations to addresses and to process and tunnel packets.

The Moving Target IPv6 Defense (MT6D) is a network layer MTD that dynamically changes IPv6 addresses mid-session while still maintaining continuous communication [3]. It is designed as a network gateway device to be placed between a protected host and the Internet, similar to a router or firewall, and should be transparent to the communicating hosts. MT6D regularly changes the source and destination addresses so that a single static address cannot be targeted for attack. Although MT6D provides strong privacy for the protected systems, significant computational resources are required to tunnel all the traffic and dynamically change network addresses.

MT6D was originally written in Python language for ease and speed of development and for portability to different system architectures. However, this implementation suffers from severe performance limitations. By rewriting MT6D in C language, the defense can work with operating system specific network application programming interfaces (APIs) and optimizations. Consequently, MT6D gains significant performance improvements and increases network throughput while operating more efficiently, allowing it to be used in constrained resource environments, such as mobile computing platforms, without introducing unacceptable delays.

1.1 Overview

This work describes the development of the C version of MT6D and was conducted in three phases. The first phase involves analysis of the code of the Python version of MT6D to understand its design as well as what areas might be performance bottlenecks. Based on this analysis and on feedback from other MT6D research efforts, specific functions of the Python version are targeted for improvement in the C version by either using functionality of the Linux kernel and network stack or by reworking the function in a more efficient way. In the second phase, information from the first phase is used to write the C version. An effort is made to use functions in C that are not specific to MT6D, but could be used in any MTD that must capture, analyze, modify, and tunnel packets. Finally, the third phase involves testing the new C version against the Python version and the performance with no MT6D as a baseline. Tests are selected and designed to show the difference in performance when adding MT6D to an existing system, as well as between Python and C, using metrics to measure network performance and host resource use.

1.2 Organization

The rest of this work is organized as follows. Chapter 2 gives some background information on IPv6, an overview of the MT6D protocol, and an analysis of the existing Python version. Chapter 3 details the development of the C version, including specific designs that were chosen and the expected effect on performance those choices would have. Chapter 4 describes the battery of network and host tests to be performed for comparing the performance of C MT6D with the Python version and without MT6D. Chapter 5 presents the results of these tests and the effect various network conditions exerted on the network and host performance of MT6D.

Finally, Chapter 6 provides conclusions on the successes and shortcomings of the work and proposes what might need to be done to improve future generations of the system.

Chapter 2: Overview of MT6D

This chapter gives an overview of the MT6D protocol on which this research is based. The first section gives some background information on IPv6, which is the primary protocol that MT6D is based on. This overview leads into a discussion of the security concerns in IPv6 that prompted the development of MT6D. The MT6D protocol itself and the way that it handles the security issues of IPv6 is covered next. Finally, the original Python implementation of MT6D, to which the C version is compared, is described.

2.1 Background

The Internet Protocol (IP) is one of the most fundamental protocols of the public Internet. The Internet Protocol defines the addressing scheme used to uniquely identify every host connected to the Internet, it describes the format of packets transmitted on the Internet, and it lays the framework that allows packets to be routed from their source to their destination. Internet Protocol version 4 (IPv4), which was first defined in September 1981 by RFC 791 [4], is the predominant version in use today, but shortcomings in its design led to the creation of its successor, IPv6.

2.2 Changes in IPv6 over IPv4

IPv6 is the primary protocol on which MT6D is based. The IPv6 protocol was originally defined in December 1995 by RFC 1883 [5] and updated in December 1998 by RFC 2460 [6]. As a successor to IPv4, IPv6 provides the same basic functionality as IPv4, but with a number of enhancements, which are briefly described in the next sections.

2.2.1 Larger Address Size

The IPv6 protocol increases the size of an Internet protocol address from 32 bits to 128 bits with the stated goal of supporting “more levels of addressing hierarchy, a much greater number of addressable nodes, and simpler auto-configuration of addresses” [6]. As an example of the increased address size, the 32 bit address size of IPv4 provides approximately 4.3 billion unique addresses, but the 128 bit address size of IPv6 provides approximately 340 undecillion.

Numerically, this increase would provide 6.2 x 10^{22} or 62 sextillion addresses per square foot of the surface of planet Earth [7].

2.2.2 Simplified Header Format

Some fields of the IPv4 header were made optional or even dropped, since most packets did not make use of these header fields. The IPv6 header is also fixed at a specific size (40 bytes) rather than being a variable size. These changes improve processing time at each hop for the most common packets.

2.2.3 Improved Option and Extension Support

In IPv6, options are no longer a part of the header, but are, instead, contained in specific sections of the packet, usually following the header and before the payload. Multiple option sections can be chained together, with the size of the option section indicated by the option size field of the option header. This design allows new options to be introduced without substantial changes to the protocol and without changes to the network stacks of deployed devices, since these devices simply can skip over options they do not understand.

2.2.4 Flow Labeling

IPv6 allows packets to be marked as belonging to a particular "flow" of packets to which special consideration should be given, such as a different quality of service. Flow labeling also can be useful in debugging by capturing only packets that are part of a specific flow.

2.2.5 Authentication and Security

The specification of IPv6 acknowledges the many privacy and security issues related to the Internet by requiring implementations of IPv6 to support the Internet Protocol security (IPsec) security architecture defined in RFC 2401 [8].

2.3 Address Assignment in IPv6

Addresses can be assigned automatically in IPv6 using Stateless Address Autoconfiguration (SLAAC) or the Dynamic Host Configuration Protocol for IPv6 (DHCPv6). While both methods may be in use at a particular site at the same time, SLAAC is typically used

where local administrators are not concerned with the specific addresses that hosts use as long as they are correct and routable; DHCPv6 is used where administrators desire more control over specific host address assignment.

2.3.1 Stateless Address Autoconfiguration (SLAAC)

SLAAC, defined in RFC 4862 [9], provides a minimalistic approach to address assignment. No additional servers are required; only a previously configured IPv6 router is necessary. Upon enabling a multicast-capable interface (usually at boot), a host begins the autoconfiguration process as shown in Figure 2.1. The first step in this process is to generate a link-local address from an interface specific identifier (usually the media access control (MAC) address) and the well-known link-local prefix defined in RFC 4291 [10]. Before this address is used on the interface, the host must verify the uniqueness of the address. The host sends a Neighbor Solicitation message to the address it has generated and awaits a response. A response indicates that the generated address is not unique. Manual intervention may be required and, unless an alternate interface identifier is available, the autoconfiguration process fails. Otherwise, the generated address is assigned to the interface. At this point, the host has IP connectivity and can communicate with other hosts on the local link. To continue the autoconfiguration process, the host transmits a Router Solicitation message to the all-routers multicast group. If no Router Advertisement messages are received, the host assumes that no IPv6 routers are present and autoconfiguration ends (although configuration by DHCPv6 may still be possible). If SLAAC is enabled on a local router, it responds with a Router Advertisement message that contains a subnet prefix and a lifetime value. The subnet prefix is combined with an interface specific identifier to form the global address to assign to the interface. The lifetime value determines how long this assigned address may be used. After checking for uniqueness of the global address, it is assigned to the interface and autoconfiguration is complete.

2.3.2 Dynamic Host Configuration Protocol for IPv6 (DHCPv6)

DHCPv6 is defined in RFC 3315 [11] as a stateful counterpart to SLAAC. DHCPv6 messages are sent using IPv6, so a link-local address or an address assigned by another method

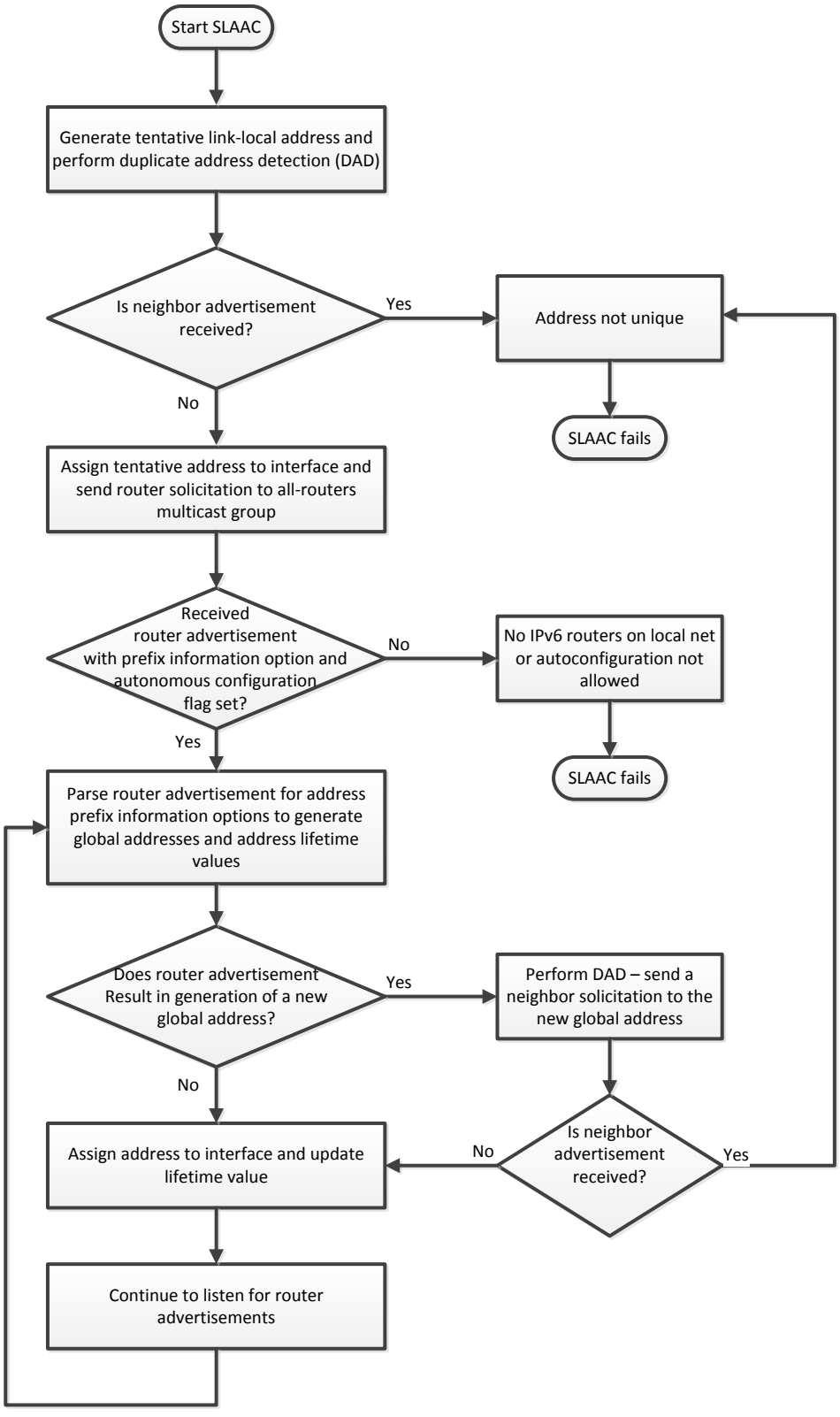

Figure 2.1 - Stateless Address Autoconfiguration Process

must be available on an interface. As shown in Figure 2.2, a client begins by sending a Solicit message to the Internet Assigned Numbers Authority (IANA) defined "DHCP relays and servers" multicast address. A DHCPv6 server, which determines that it can service the client's request, responds with an Advertise message. The client picks one of the servers that responded and replies to it with a Request message. The server then replies with a Reply message, which confirms the assignment of an address and provides related configuration information to the client. Once the client receives the Reply message, it may begin using the assigned address and the DHCPv6 exchange is complete.

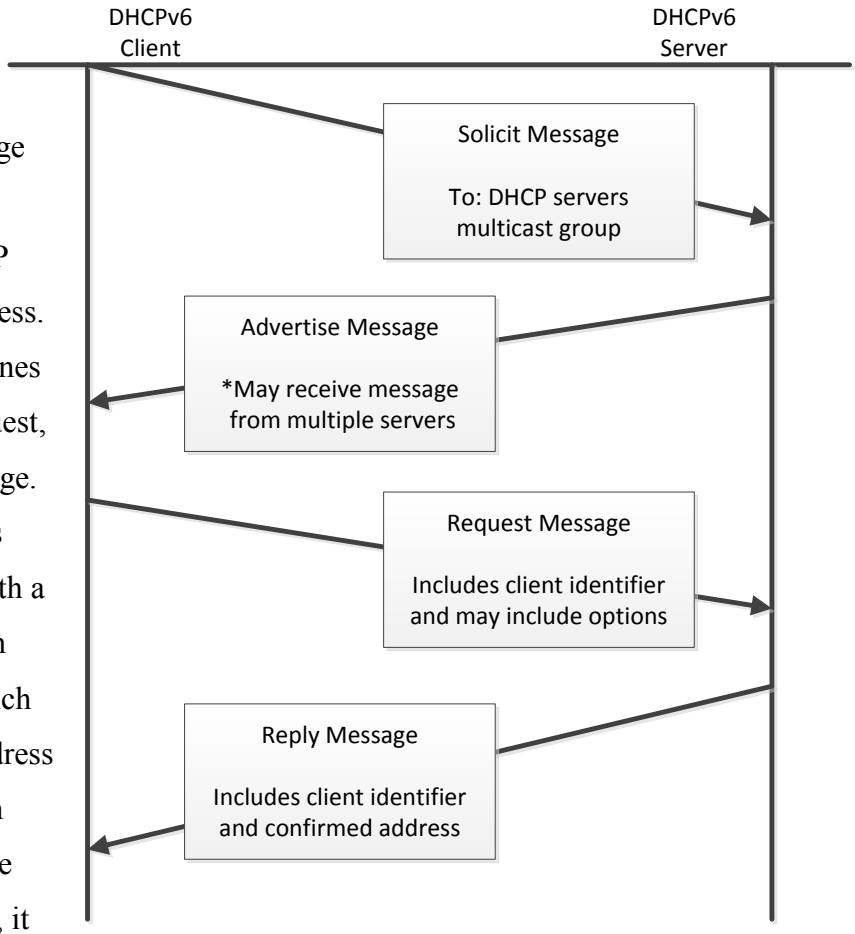

Figure 2.2 - Dynamic Host Configuration Protocol Exchange

2.4 Motivation for MT6D

The two address assignment methods discussed previously (SLAAC and DHCPv6) both use identifiers that are unique to the interface and/or host that could potentially be tracked, creating privacy and security concerns [12]. In SLAAC, the interface identifier is typically generated from the globally unique interface MAC address. The interface identifier becomes part of the global address, which is visible when the host connects to other systems. This design is especially concerning for mobile hosts, such as smart phones and tablets. In the example scenario shown in Figure 2.3, an attacker could write a malicious app that, once installed on the target host, would repeatedly ping the attacker's systems. The attacker could monitor the movement of the host by recording pings that originate from an address with the interface

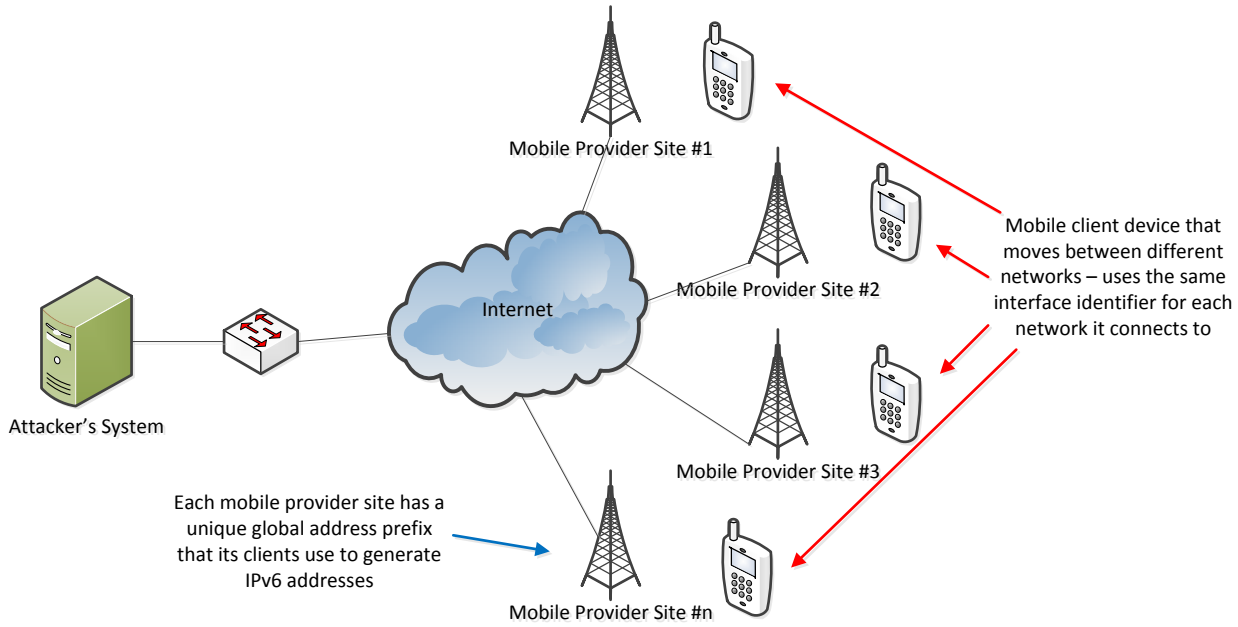

Figure 2.3 - An Attacker Tracking a Mobile Client

identifier of the target host and using the network prefix portion of the address to determine the geographic location of the host.

In DHCPv6, a Dynamic Host Configuration Protocol (DHCP) unique identifier (DUID) is used by the DHCPv6 server to maintain state about its clients. This DUID value is maintained by the client and included in messages to DHCPv6 servers to facilitate a consistent state. Though the specification allows the DUID value to change, it is generally not changed since doing so would interrupt the stateful design. DHCP traffic would typically not be visible outside the link-local network, but an attacker could install a DHCP relay at specific sites to forward traffic out of the network. By monitoring this traffic and making note of the DUID, an attacker could track the movements of hosts between networks [13].

To combat these vulnerabilities, MT6D was designed to hide a host's true IPv6 address from an attacker by regularly generating new addresses for the hosts it protects, hiding the original addresses. MT6D takes advantage of the large address space of IPv6 by continuously changing addresses, creating a moving target that is computationally difficult to reacquire.

2.5 Moving Target IPv6 Defense (MT6D)

The purpose of MT6D is to allow two or more hosts to communicate over a public

network without divulging their individual identities. MT6D accomplishes this by greatly increasing the apparent number of active addresses that an attacker might observe, making correlation of communication with specific hosts extremely difficult. The methods used by MT6D include dynamic obscuration of IPv6 addresses and transmitting protected traffic inside an encrypted (or unencrypted) tunnel [14].

2.5.1 Session Key

The MT6D implementation maintains a session key for each set of two hosts that are communicating through it. The session key is used for encrypting tunneled packets, if enabled, and for generating IPv6 addresses, as described below[14]. The size of the session key is implementation dependent; the reference implementation of this current research uses a 512 bit session key. The two ends of an MT6D stream renegotiate the session key at a set interval, making it more difficult for an attacker to determine the key. The present implementation requires that keys be synchronized at both ends out-of-band before the MT6D implementation is started. This parameter could be changed to use a well-defined key exchange method, such as Diffie-Hellman [15].

2.5.2 Address Hashing

To implement dynamic obscuration of IPv6 addresses, MT6D implementation maintains an alternate set of IPv6 addresses, referred to hereafter as “hashed addresses” [14]. As described later, the original source and destination addresses of a packet (true addresses) are removed by MT6D implementation and the packet is tunneled inside an MT6D packet, which uses the hashed addresses for its source and destination. The hashed addresses are regenerated often, according to the parameters of MT6D implementation. To generate a new hashed address, a hashed address calculation (Equation 2.1) is used where a hash is taken of the host portion of the original (source or destination) IPv6 address, the session key, a constantly changing value (such as the current time), and an interface specific identifier (such as a MAC address). The result is truncated and added to the original host portion of the address to create a new IPv6 address. For each two hosts that are communicating, a source and destination address are created. A properly configured and synchronized MT6D implementation can create a matching set of source and

destination addresses at both ends of the connection. These addresses are used to communicate across the public network.

$$\text{trueAddress}[0-63\text{bits}] + \text{hashFunction}(\text{trueAddress}[64-127\text{bits}] + \text{sessionKey} + \text{salt})[0-63\text{bits}]$$

Equation 2.1 - Hashed Address Calculation

2.6 MT6D Protocol

Communications using MT6D involve two end hosts that want to exchange information across a public network where security and/or privacy is a concern. In order to protect traffic between the two hosts, an implementation of MT6D must be located between each host and the border of the public network. MT6D is designed to operate either on the individual hosts or on a gateway device that is located at the edge of a trusted network before transition to the public network [14]. Figure 2.4 shows these two operating modes.

Whether on the host itself or on a gateway device, MT6D implementation intercepts all packets that are passing through it and examines each one. If a packet is not part of an MT6D protected stream, it is allowed to pass unchanged. Otherwise, the packet is modified and becomes part of the MT6D tunnel [14].

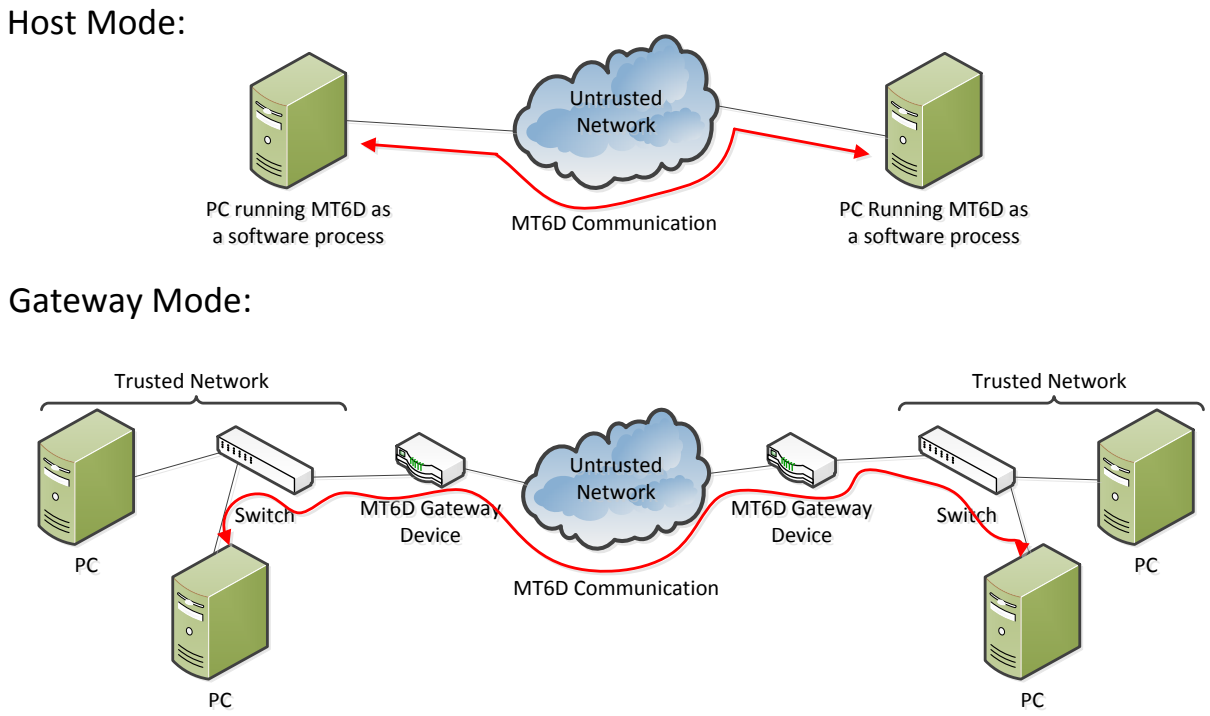

Figure 2.4 - MT6D Operating Modes

When a packet is tunneled, it is placed inside an MT6D packet for transmission across the public network. The Ethernet header and checksum is regenerated by the MT6D implementation at the other end, so they are discarded. Most of the IPv6 packet is directly copied into the payload of the MT6D packet, with the exception of IPv6 addresses. Since one of the major goals of MT6D is to prevent address correlation, original IPv6 source and destination addresses are removed so that the MT6D packet's payload consists of the first 8 bytes of the original packet's header, followed immediately by the original packet's payload. Original IPv6 source and destination addresses are added back to the packet by MT6D implementation at the other end. Figure 2.5 shows how the tunneled packet is constructed from the original packet.

The MT6D packet uses the current set of hashed IPv6 source and destination addresses. If encryption is enabled, the MT6D packet's payload, including the original packet's remaining header fields and payload, is encrypted prior to transmission. Once this process is complete, the packet is sent through the public network to MT6D implementation on the other side.

Upon receiving a tunneled packet from the public network, MT6D implementation checks whether the packet is part of a valid, ongoing communication between two hosts by attempting to match the packet's source and destination addresses with hashed addresses it recognizes. If no match is found, the packet is forwarded with no modifications. If a match is found, the packet is processed by the MT6D implementation. Hashed source and destination IPv6 addresses used on the packet are used to identify the MT6D stream to which the packet belongs. The packet payload is extracted as shown in Figure 2.6 and, if enabled, decrypted. The IPv6 header in the extracted payload has the original IPv6 source and destination addresses added back and a new Ethernet frame is generated before sending the packet on to its final destination.

2.6.1 Overhead

By tunneling packets inside new IPv6 packets, MT6D introduces a certain amount of overhead to every packet it handles. MT6D uses the user datagram protocol (UDP) as the transport layer protocol to carry tunneled packets, which results in an increase of 40 bytes for the additional IPv6 header [6] and 8 bytes for the UDP header [16], for a total of 48 bytes of overhead [14]. The Ethernet frame is ignored since it is stripped off the packet before tunneling

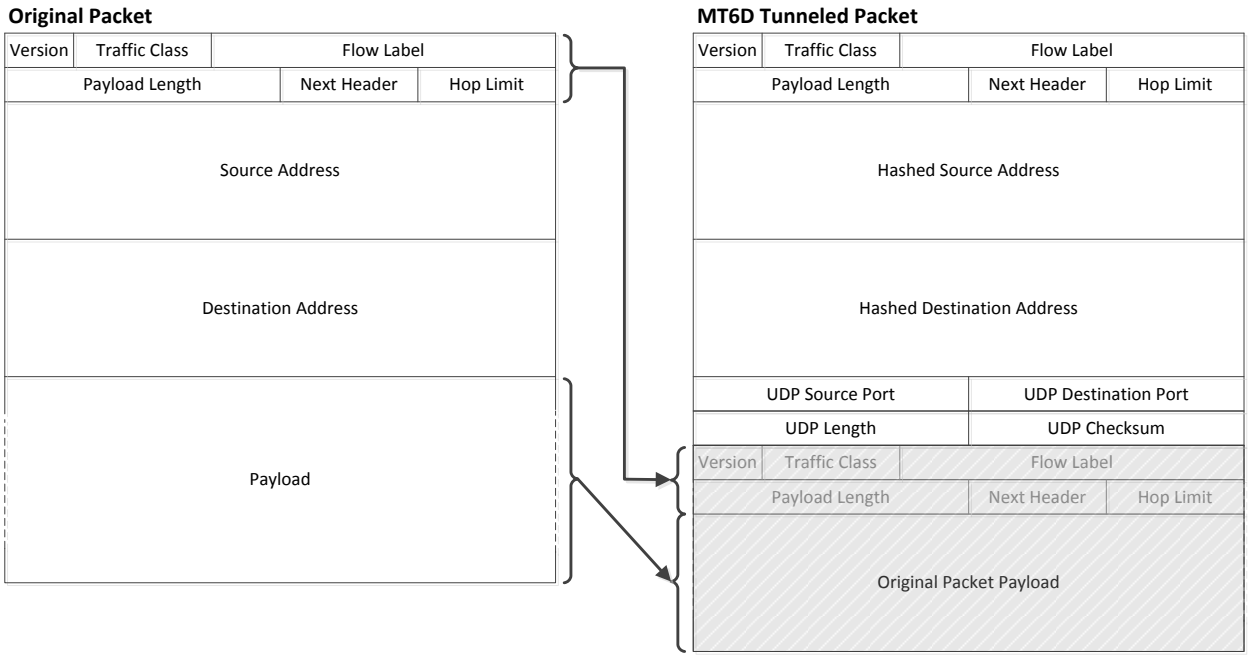

Figure 2.5 - Original Packet Fields in Tunneled Packet

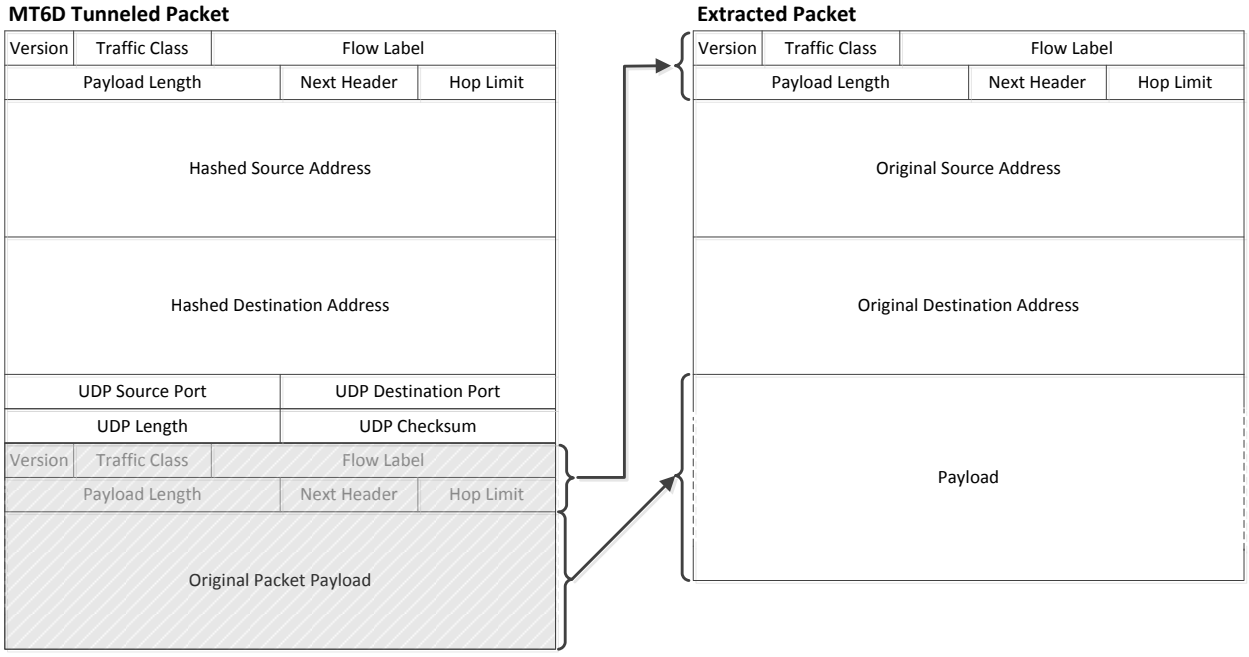

Figure 2.6 - Tunneled Packet Fields in Extracted Packet

and added back after tunneling. The 48 bytes of overhead is reduced by removing the original packet's IPv6 addresses from the payload. Each IPv6 address is 16 bytes; removing both the source and destination address lowers the total overhead to a final 16 bytes. The overhead may be increased if certain MT6D optional features are used. For example, packet encryption requires the use of an IPv6 destination option on the tunneled packets to indicate to the other side that encryption is in use – this information adds 8 bytes to the overhead. The total real-time overhead is taken into account by the Maximum Transmission Unit (MTU) that MT6D advertises so that "Packet Too Big" errors are not generated.

2.6.2 Neighbor Discovery Protocol (NDP) Packet Handling

Neighbor Discovery Protocol (NDP) [17] packets are handled as necessary to support the operations of MT6D. Devices on the network may send multicast NDP messages, but, as these are not host specific, they do not require modification. Messages such as router solicitations required by SLAAC are altered to use the current hashed address before being forwarded. Neighbor solicitation messages (used either for duplicate address detection or to determine another host's MAC address) to a host that is protected by MT6D or to an active hashed address in use by MT6D is dropped and responses generated that contain the MT6D implementation's information, rather than the target host. This causes packets destined for a protected host or hashed address to be addressed to the MT6D implementation's link-layer address. Figure 2.7 shows this process.

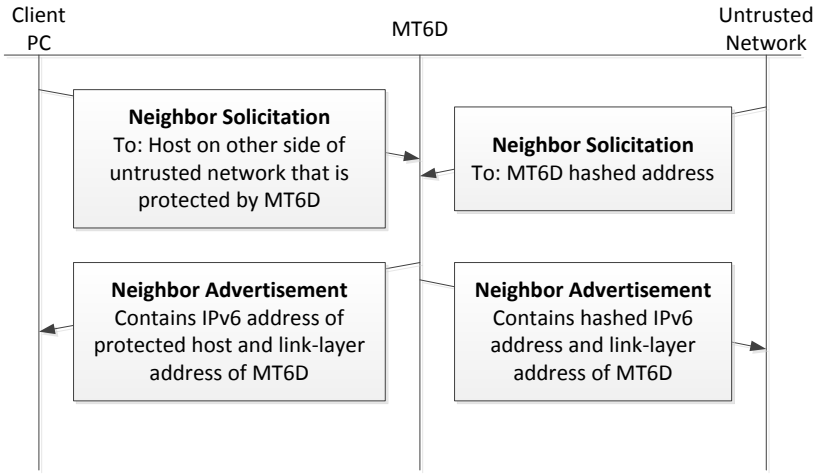

Figure 2.7 - MT6D Handling of NDP Packets

2.6.3 Internet Control Message Protocol (ICMP) Packet Handling

Internet Control Message Protocol (ICMP) for IPv6 (ICMPv6) [18] packets may be generated by hosts on either end of the MT6D stream, and are tunneled through MT6D like any other packet. However, ICMP messages generated by intermediate devices in the path between

two MT6D implementations require special handling. These ICMP messages typically report a communication error and contain as much of the offending packet in their payload as can fit. The MT6D implementation extracts this payload, decrypts the original packet if necessary, and replaces any hashed addresses with original addresses. A new ICMP packet of the same type as the one received is generated, using the modified payload from the original (Figure 2.8). In the specific case where the ICMP packet is a "Packet Too Big" message, the advertised MTU is reduced by the MT6D overhead. The new ICMP packet is then forwarded to the host.

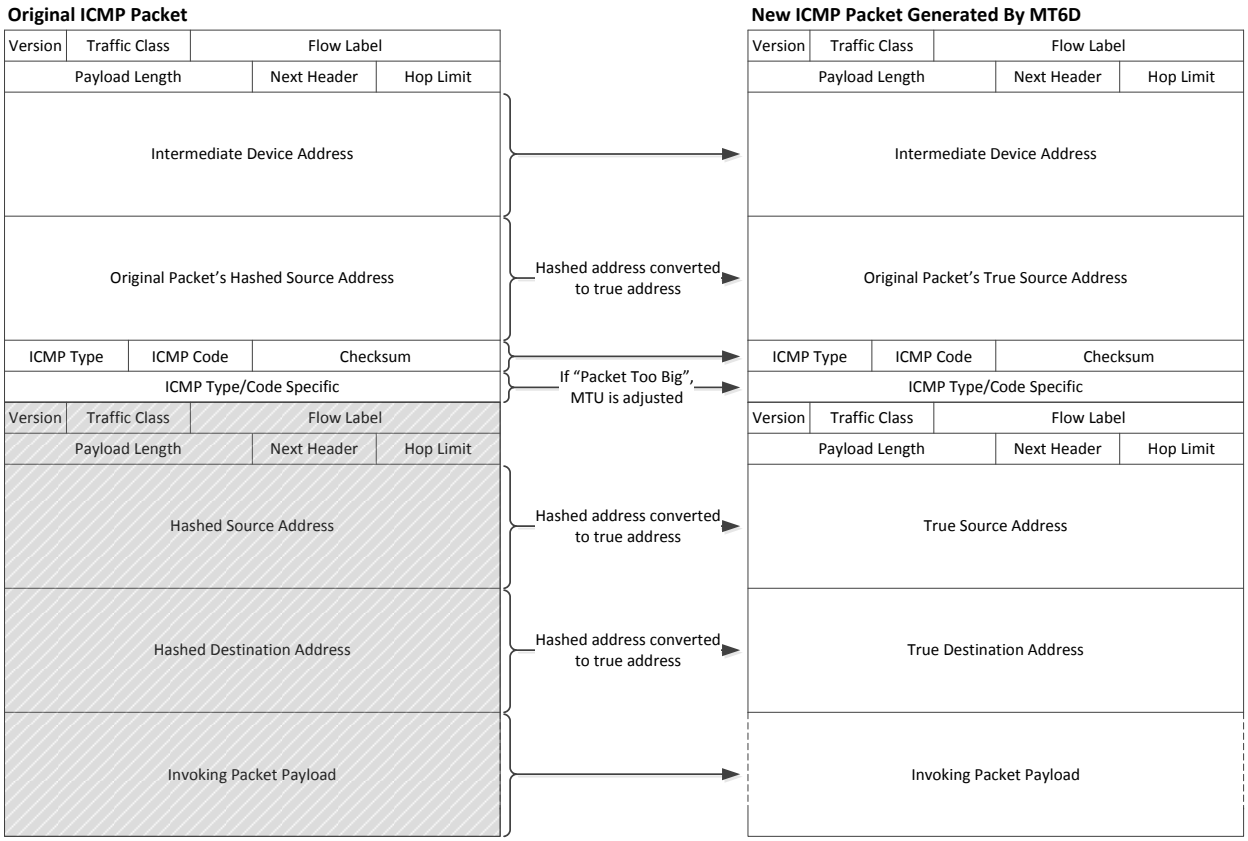

Figure 2.8 - MT6D ICMP Packet Handling

2.7 An Implementation of MT6D in Python

As a means of validation and testing, a working implementation of MT6D is necessary. The Python language was chosen for the implementation as a rapid method to produce a working prototype. This section examines Python implementation, how it works, and some issues encountered with it.

2.7.1 Overview of Python Implementation

The Python implementation captures packets going through it, determines if they are part of a protected MT6D communications stream, and tunnels them accordingly. Packets are not actually modified; instead, matching packets are dropped and new packets are generated for forwarding. Packets are captured using the Python interface to libpcap [19], and packets are dropped by inserting relevant rules into the ip6tables firewall [20]. Python implementation drops and generates packets by using three threads – a main (startup) thread, a packet listener thread, and a rehash worker thread.

2.7.2 Main Thread

The main thread starts the program and initializes various data structures. It begins by parsing the command line options, parsing the main configuration file, and setting up logging. Then, the three auxiliary configuration files are parsed. The first, the users file, contains hostname to IPv6 address mappings. The second, the ethers file, contains IPv6 address to MAC address mappings. The third, the profiles file, contains statements that describe an end-to-end MT6D communications stream by listing the source host, the destination host, and the session key that the two hosts have agreed to use.

Next, the firewall is initialized. Python implementation uses ipsets [21] as a more efficient way to facilitate constant address changes that are needed as IPv6 addresses are rehashed. At startup, a new ipset is created for storing IPv6 addresses by calling the ipset executable. Then rules are added to ip6tables to drop any packets being forwarded through the host with a source or destination address that is listed in the ipset. The last step of the firewall initialization is to add rules that drop any neighbor/router solicitation/advertisement packets being forwarded, as these packets are also changed by MT6D.

After initializing the firewall, the key storage is initialized, which uses a SQLite3 [22] file to store session keys and related attributes. The route storage is also initialized, which uses the same SQLite3 file to store information about active routes. Python implementation uses routes to refer to a mapping between the true host addresses and hashed addresses.

Finally, the main thread starts the packet listener thread and the rehash worker thread. It then idles while waiting for a keyboard interrupt, at which point it cleans up the other threads and terminates the program.

2.7.3 Packet Listener Thread

The job of the packet listener thread is to receive packets from the network and send out transformed copies of the received packets (Figure 2.9). On startup, the packet listener thread opens the forwarding network interface with libpcap and applies a filter such that all IPv6 packets are captured. Once a packet is received, it is examined; the resulting flow depends on the packet type, with router solicitations and advertisements handled first. For a router advertisement, the MTU contained in the packet is adjusted to account for the overhead of MT6D; then the packet is forwarded on to the protected hosts. Router solicitations are modified so that no identifying information about the protected host is leaked to the public network by setting the source IPv6 address in the IP header to the unspecified address and the source MAC address in the Ethernet header to a random value. These modifications are copied into new packets and forwarded to their destinations.

If the received packet is not a router solicitation or advertisement, the packet listener thread looks up the route associated with the packet using the IPv6 source and destination address. If no corresponding route is found, the packet is ignored. If the packet does match a known route, the route that is found (based on the IPv6 addresses) indicates the direction of the packet. A packet containing hashed addresses indicates that the packet originated from the public network and should be extracted from its tunnel and forwarded to the end host. A packet containing true addresses indicates that it was received from a protected host and should be tunneled before being forwarded through the public network.

After looking up the route, all remaining ICMPv6 packet types are handled. For neighbor solicitations, a new neighbor advertisement packet is generated with true IPv6 addresses, and MAC addresses are resolved from the internal “IPv6 address to MAC address” table. A “Packet Too Big” packet has its MTU adjusted by the MT6D overhead. Finally, IPv6 addresses in the packet payload (which contains a partial copy of the invoking packet) are replaced with route addresses. The packet is then sent to the protected host.

At this point, only normal (non ICMPv6) packets are left to handle. The route determines if the packet is destined for the protected host or the public network. Packets destined for the public network are tunneled into the payload of a new IPv6/UDP packet using hashed addresses. If encryption is enabled, the payload is encrypted, the required destination option header

describing the encryption type is added to the IPv6 packet, and the packet is sent. Packets destined for the protected host are extracted from the tunneled packet, decrypted if necessary, and the true IPv6 addresses are reinserted before being sent.

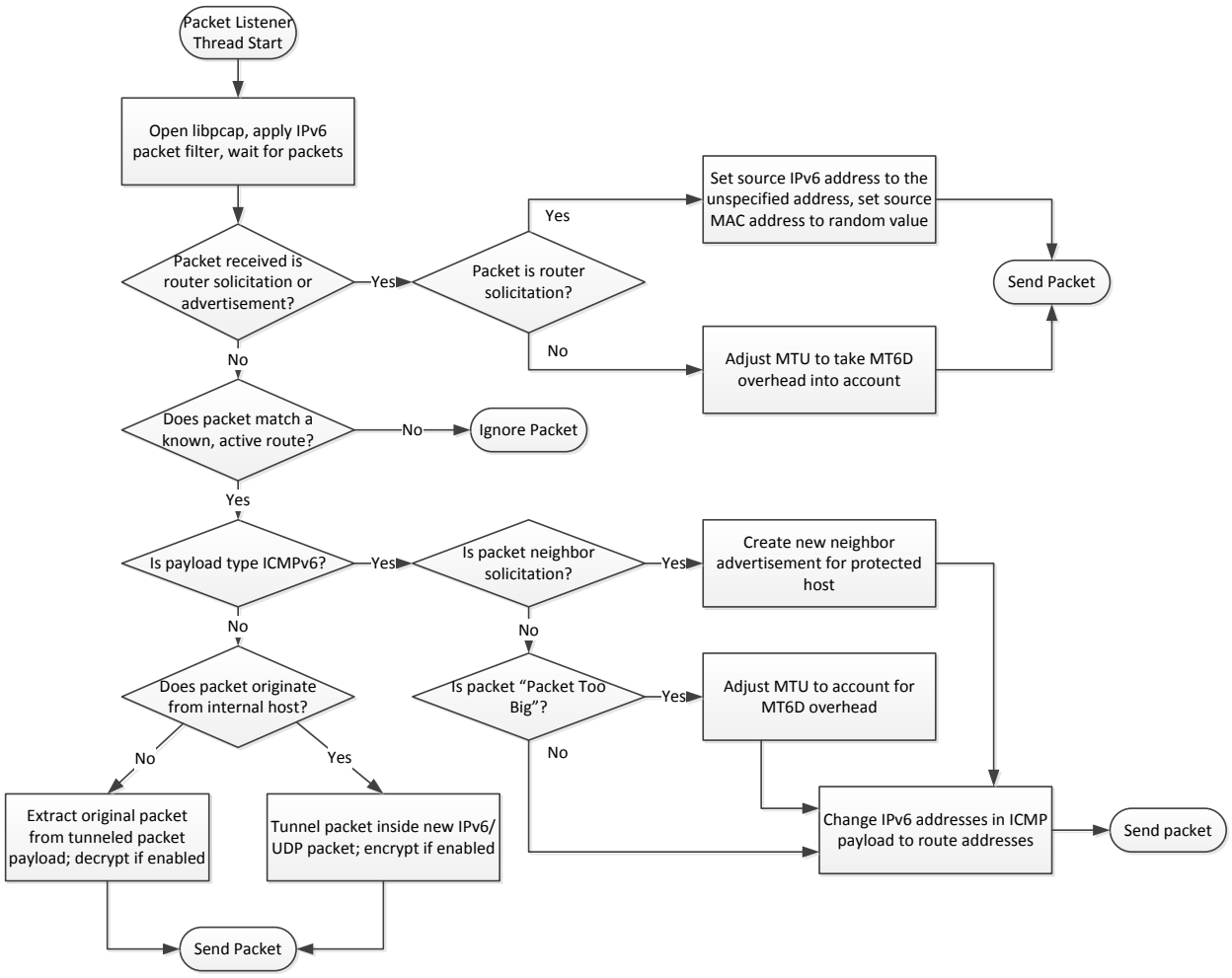

Figure 2.9 - Python MT6D Packet Listener Thread Flow

2.7.4 Rehash Worker Thread

The rehash worker thread is implemented as a repeating loop, as shown in Figure 2.10. The frequency of the loop is the configured rotation time of the hashed addresses, as each iteration of the loop generates new hashed addresses. The loop begins by cleaning up any expired hashed addresses and their associated routes. Hashed addresses expire after a certain lifetime as set in the configuration. The address rotation time and the address lifetime should be

set so that at least one current hashed address is always present in a profile. Once the cleanup is complete, new hashed addresses are generated in a process called profile mutation. The secure hash algorithm (SHA)-256 [15] algorithm is used to generate a hash of the host portion of an IPv6 address (the last 64 bits), the profile session key, and a salt value (based on the current time). The first 64 bits of the hash are appended to the end of the first 64 bits of the true IPv6 address to create a new hashed IPv6 address. Two of these addresses are created, a source and a destination, and stored in a profile object along with the session key. This process is completed for each active profile in the MT6D configuration.

Once new profiles are generated, the profiles are bound. The source address of the profile is bound to the network interface of the MT6D host. The source address is also added to the ipset used in ip6tables so that packets received to that address are dropped rather than forwarded. The new profiles are then stored in a local profile storage object with their calculated lifetimes.

Finally, two new routes (an internal and external route) are created for each mutated profile. Because the packet listener thread uses hashes to look up a route when it receives a packet, hashes are used as a key to identify the routes. The hashes are based on the current salt and the IPv6 addresses of the related internal or external route. The internal route is created first, followed by the external route. For the internal route, a new route is created and stored consisting of a hash, the true destination IPv6 address, and the true source IPv6 address. The internal route's hash is created from the external route's destination IPv6 address, source IPv6 address, and salt. Conversely, the new external route is created and stored and consists of a hash, the hashed source IPv6 address, and the hashed destination IPv6 address. The external route's

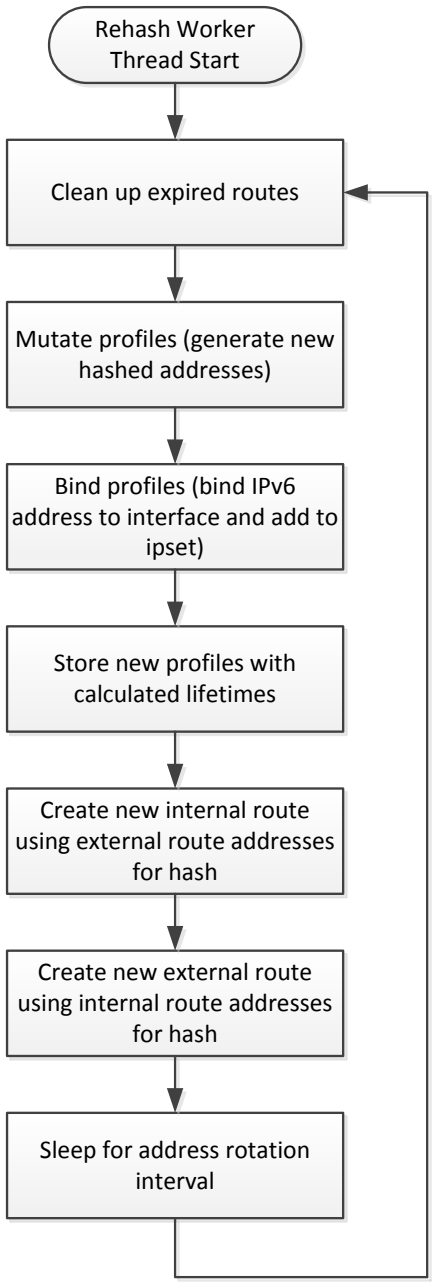

Figure 2.10 - Python MT6D Rehash Worker Thread Flow

hash is created from the internal route's source IPv6 address, destination IPv6 address, and salt. These two interlocking routes allow the packet listener thread to look up the internal route when it receives a packet with external (hashed) addresses and to look up the external route when it receives a packet with internal (true) addresses.

2.8 Python Implementation Performance Concerns

Initial testing of the configuration revealed room for improvement in the performance of the Python implementation of MT6D. The major concern was the use of libpcap to capture packets and the interaction of libpcap with Python's global interpreter lock (GIL) [23]. GIL is a mutex that prevents multiple native Python threads from executing Python code at once, due to the thread-unsafe nature of the memory management design. The result is that, when a packet is received on an interface, libpcap processing is necessary to capture the packet, which prevents other Python threads from running, resulting in decreased performance. Another concern was the performance hit taken by the original design of polling for the proper time to rehash addresses. In the version of Python MT6D studied, this aspect had been changed to a simple sleep for the rotation interval. However, this design requires that clocks on each MT6D device are held in tight synchronization by another means (such as the Network Time Protocol (NTP)) [24]; otherwise, different hashed addresses are generated by each MT6D implementation, which leaves the hosts unable to communicate.

Results of performance testing of the Python implementation are given in a later chapter. Initial concerns led to the decision to create another implementation of MT6D in a more efficient language. The next chapter describes an implementation in C created for this research, the design decisions made, and the similarities and differences with the Python version.

Chapter 3: Design of C MT6D

The C version of MT6D had several design goals, mostly for comparison with the Python version. After analyzing the design of the Python version, specific areas were noted that could be redesigned or implemented differently in the C version, with the ultimate goal of increased performance and efficiency. Some decisions made while implementing the C version were influenced by the design of the Python version and stand out as uncommon methods of designing a C network program. This chapter describes the design goals of the C version, the differences between it and the Python version, and the specific design of the C version, including the threading model and program flow. For the source code of the C version, see Appendix A.

3.1 Design Goals and Version Differences

Most design goals of the C version emerged from examination of the Python version for areas that could be improved in terms of performance and efficiency. A requirement, however, was to create a version of MT6D in C that would be compatible with the Python version. Specifically, creating a compatible version required an installation of the C version running on one host and an installation of the Python version running on another host to be able to communicate with each other once the configuration files are synchronized. Achieving such a system required that the C version implement the same features and calculate IPv6 addresses in the same way as the Python version. The assumption was that simply implementing MT6D in C would realize some performance and efficiency increases since C is a compiled language and Python is an interpreted language [25].

In addition to the assumed gains from the C version, specific areas of the Python program were targeted for improvement in the C version. One of the most important areas identified was the method by which the Python version receives packets from the network. As discussed in Chapter 2, the Python version uses libpcap to capture packets and iptables rules to drop the same packets, effectively "receiving" them. The typical way to receive packets in a program, by listening on a network socket, is not adequate for MT6D since it must intercept all packets (regardless of port or protocol) destined for a specific host. Due to the problems mentioned in Chapter 2 with libpcap and Python's GIL, using libpcap to receive packets results in a major

performance decrease in the Python version. This disadvantage was taken into account when designing the C version, which led to the decision to use Netfilter queues [26] instead of libpcap to receive packets.

Other differences between the versions include an attempt to maximize the amount of work performed by the Linux kernel, instead of the program itself. For example, rather than MT6D capturing NDP packets to remote hosts and crafting responses to them, the C version binds the remote host's address to the local host, so that NDP packets are received and processed by the networking stack. The C version also uses more threads to leverage a host's multiprocessing capabilities – at least four threads always run, with an additional thread for each additional set of hosts communicating. Locks are also avoided; instead, atomic operations are used when data that are shared by multiple threads must be modified [27]. Finally, unlike the Python version, the C version does not need to call any external programs (such as ip, or iptables) as it uses functionality from other libraries.

3.1.1 External Libraries

The C MT6D program uses several external libraries to provide required functionality. These libraries are used so that calls to external programs are not needed, providing an efficiency increase. Libraries used are crypto & ssl [28], netfilter_queue, ip6tc & xtables [29], and netlink [30]. Crypto and ssl libraries are included to access cryptographic features of the OpenSSL toolkit. Specifically, MT6D uses the hashing and encryption functions. The exact algorithms used can be configured by the user, but the default is to use the SHA hash function with a 256 bit digest and the Advanced Encryption Standard (AES) encryption algorithm [15] with a 256 bit key size in Cipher-Block Chaining (CBC) mode. The hash function is used to generate new IPv6 addresses by creating a hash of the true IPv6 address, the session key, MAC address, and a salt. The hash function output is used as the host portion of the new IPv6 address. If packet encryption is enabled in the MT6D configuration, then the encryption algorithm is used to encrypt and decrypt all packets in the stream. Otherwise, encryption is only used to protect session key exchanges between two MT6D devices.

The netfilter_queue library provides functions that allow the MT6D program to interact with the userspace queue functionality of the netfilter infrastructure in the Linux kernel. This design allows for definition of ip6tables rules that direct all packets matching the rule into the

queue. The kernel copies such packets to userspace and places them into a queue, identified by an integer that a program can wait on. When a packet is placed into the queue, the program is awakened and can access the full packet data. After processing a packet, the program can then inform the kernel if it should allow the packet to continue traversing ip6tables chains or drop the packet. The MT6D program uses netfilter queues to receive packets by inserting relevant ip6tables rules to direct stream packets into specific queues, which are read by the stream threads and icmp thread. Figure 3.1 shows how packets are directed into queues by ip6tables rules.

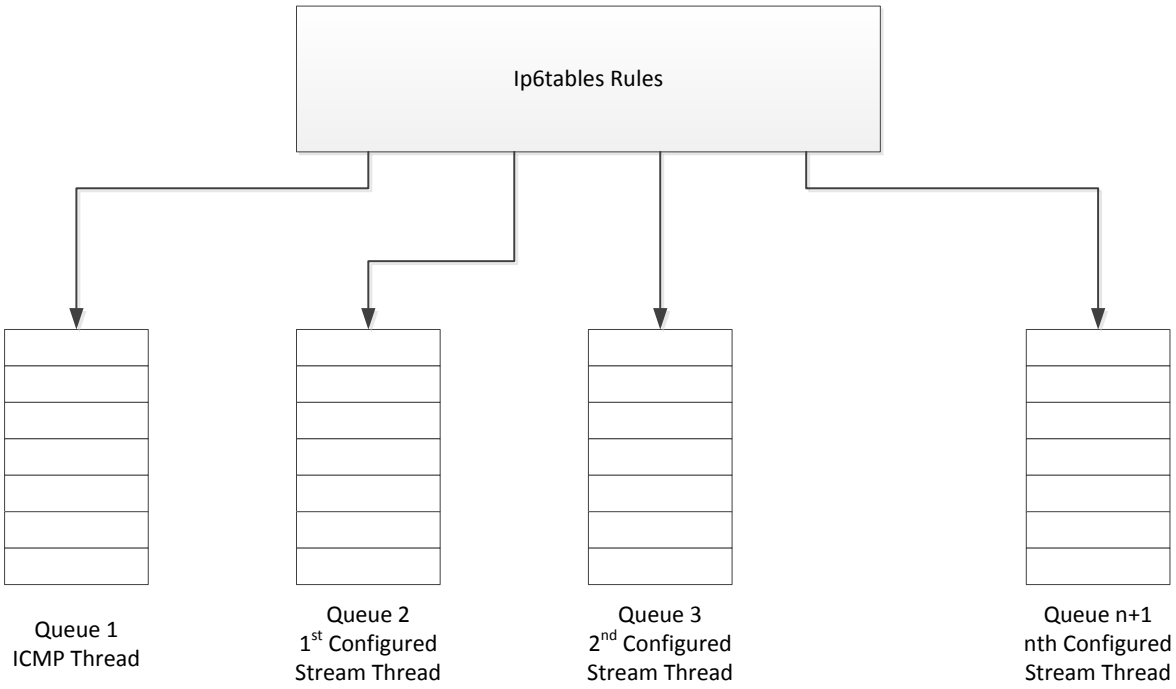

Figure 3.1 - Netfilter Queues

The ip6tc and xtables libraries together give MT6D the ability to construct ip6tables firewall rule structures and insert or remove them from the kernel. The ip6tc library defines the functions and structures necessary for creating the actual rules, and the xtables library allows the use of extended ip6tables target modules beyond the standard ones (such as ACCEPT and DROP), which includes the userspace queues. The ip6tables rules are manipulated by MT6D whenever the rehash thread generates new addresses. New rules must be inserted into ip6tables to direct packets to or from the new addresses to the appropriate stream queue; expired rules must be removed.

The netlink library is used to manipulate the network stack and has similar operations as the ip program. The library defines the necessary functions and structures to allow the MT6D program to send messages to the kernel that describe changes that should be made. Some operations that can be performed using the netlink library include adding, removing, and modifying network interfaces; binding and unbinding network addresses on interfaces; defining routes; and changing entries in the neighbor cache table. The MT6D program uses the netlink library to bind and unbind IPv6 addresses from network interfaces. On program startup, the true address of the remote host is bound to the internal network interface on the MT6D device, and each hashed source address is bound to the external network interface as soon as the address is generated. Binding the addresses in this manner is more efficient and removes some of the load from the MT6D program and places it on the kernel. For example, when the local host initially tries to send a packet to the remote host's true address, the kernel on the MT6D device responds to the host with an NDP packet since the remote host's address is bound and considered local on the MT6D device. This configuration eliminates the MT6D program from crafting NDP packets itself.

3.1.2 Streams and Routes

The C version uses the concepts of streams and routes in a different way than the Python version. In the C version, a stream refers to the total, end-to-end communication between two hosts that are behind MT6D, with the presence of MT6D being transparent to the hosts. Figure 3.2 shows a typical setup. The two hosts are communicating with each other and are separated by the Internet. Their communications pass through two MT6D devices, one at each border with the Internet, which protect the hosts' traffic inside an MT6D tunnel. The operation of MT6D is transparent to the hosts, and the hosts' operators may or may not be aware that MT6D is in use. The large outer path in the figure is the "stream," as the term is used in C MT6D. From the hosts' point of view, the stream is just the path from one host to the other. From the MT6D device point of view, the stream describes communications between two specific hosts, with a specific session key and using time dependent, hashed addresses. The stream is not specific to the protocol or direction of traffic carried by MT6D.

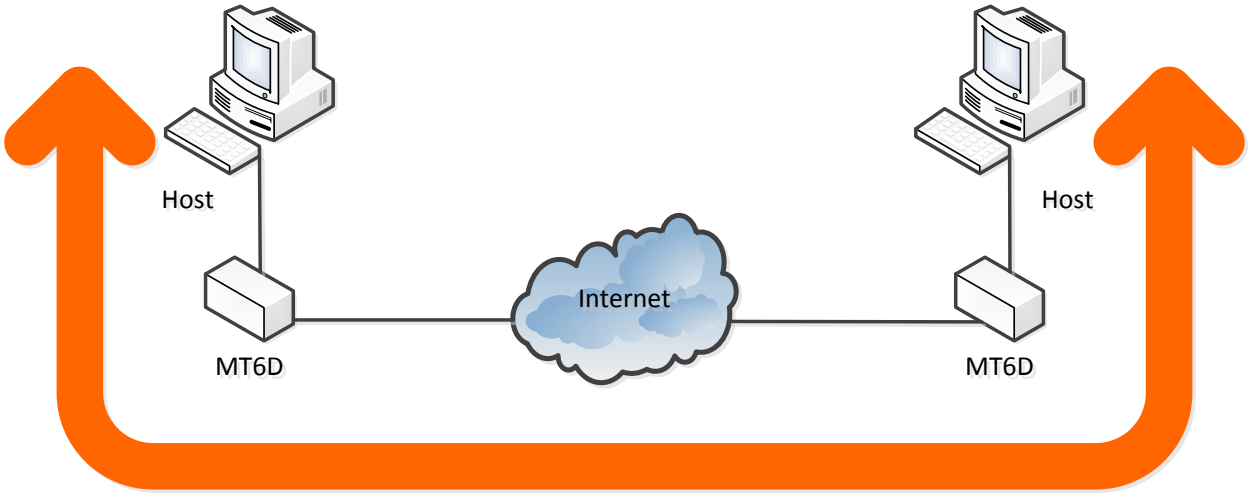

Figure 3.2 - C MT6D Stream

Closely related to a stream is a route. In C MT6D, a route comprises a hashed set of IPv6 source and destination addresses, as well as an expiration time when the route can no longer be used. Routes are created by the MT6D program at the interval defined in the configuration using true addresses, session key, MAC address, and current time to create new hashed addresses. A stream has several such routes associated with it, one which is active and the rest still valid but approaching expiration. Source and destination addresses on an incoming packet are used to match a packet to a route and, in turn, to a stream.

In C MT6D, a simple linked list of stream structures is maintained. The structure contains the thread ID of the stream thread associated with the stream structure and the netfilter queue ID where the stream's packets are directed. It contains source and destination IP and MAC addresses, the current session key, and information related to session key changing, such as a new, temporary key, key expiration time, and key exchange state. The stream structure also contains several route related pointers, as described below. Routes associated with each stream are described by structures and stored in linked lists specific to each stream. The route structures contain the hashed source and destination IPv6 addresses and the time that the route expires.

Each stream structure contains three route related pointers: a head pointer, an active pointer, and an expired pointer. Figure 3.3 illustrates the three pointers and their target lists. Route structures are stored in an ordered linked list with the newest route at the head of the list and the route closest to expiration at the tail of the list. The head pointer in the stream structure

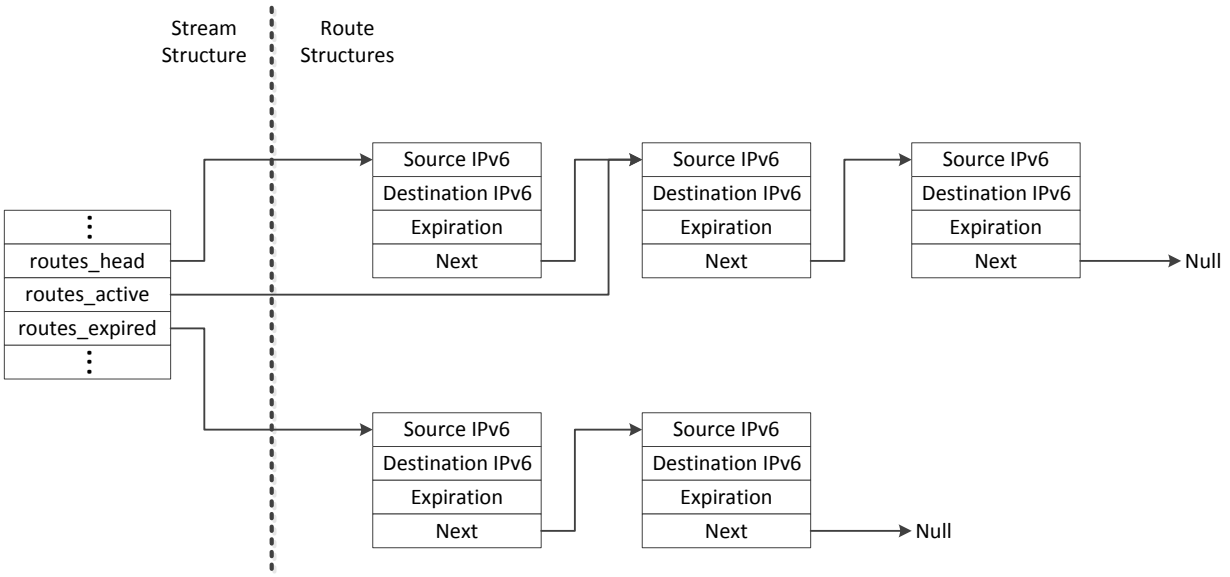

Figure 3.3 - C MT6D Stream and Route Structures

points to the newest route at the head of the list and the active pointer points to the second route immediately following the first route. Due to the delay between binding a new IP address to an interface and initial use of that address to send packets, C MT6D always uses the second newest route when sending a packet. The separate active pointer is maintained so that the stream thread is not required to go through two pointers when sending a packet. When a new route is created, the active route pointer is updated to point to the first route and the head pointer is updated to point to the new route. Both of these updates are performed using atomic operations (compare and swap) so that locking is not needed.

A separate, temporary list of expired routes is maintained while pending deletion, similar in concept to how some interpreted languages perform garbage collection [31]. The expired pointer points to this list. When an expired route is found, it is assumed that all following routes in the list have also expired (since they are stored in expiration order). The expired pointer is set to point to the first expired route and the next pointer in the previous (unexpired) route is atomically set to null, thus splitting the list. Once the list is split, no stream thread can iterate into the expired list, but a stream thread could possibly have already been in the expired part of the list before the list was split. Due to this possibility, the expired list is not immediately deleted. Instead, it is saved, with the expired pointer pointing to it until the next check for expired routes. Previously expired routes are deleted before searching for new expired routes.

Maintaining a list of expired routes, rather than deleting them immediately, avoids synchronization issues and the need for locks in the multi-threaded environment.

3.2 Threading Model

The C MT6D program consists of four or more threads that control various aspects of the program's operations. The main thread handles initialization routines and listens for signals. The rehash thread maintains the streams and routes by removing expired routes and generating new ones. The ICMP thread handles ICMPv6 packets that are outside a stream. One or more stream threads handle the packets within a stream. Each of these threads is discussed in detail in the following sections.

3.2.1 Main Thread

The main thread runs when the program starts and handles initialization and housekeeping duties, which is illustrated in Figure 3.4. The thread begins by initializing a blank configuration structure and setting some default values. The most important default value is the location of the configuration file, which is used if the user does not specify a location. The configuration structure is a memory object that is shared by all threads that contains settings defining how the program should operate. This structure is not protected by any synchronization mechanism (such as locks) since it is only modified by the main thread before any other threads have been started. Once the other threads have started, the configuration structure is treated as read-only.

Next, the main thread loads the configuration file, either from the default location or from the location specified on the command

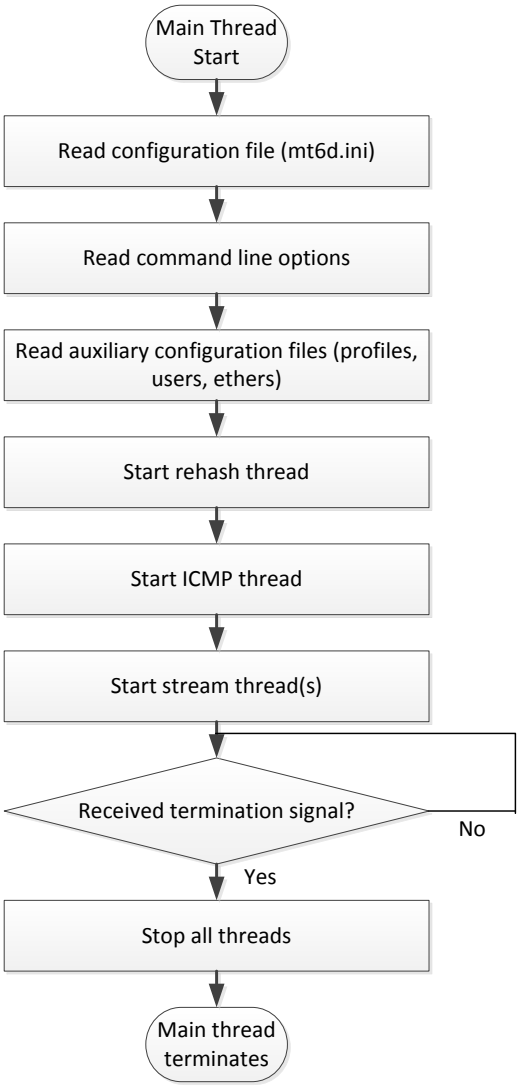

Figure 3.4 - C MT6D Main Thread Flow

line. This process fills the configuration structure with settings from the configuration file. See Table 3.1 for a description of settings stored in the configuration structure. The rest of the command line is read last so that settings specified on the command line override ones specified in the configuration file. The main thread then reads the auxiliary configuration files containing stream definitions (consisting of source and destination hostnames and a session key), hostname to IP address mappings, and IP address to MAC address mappings. At this point, all necessary configuration information has been read in.

Structure Entry	Description
bridging_nic	Network interface to bind route addresses to
internal_nic	Network interface connected to protected hosts
external_nic	Network interface connected to Internet
rotation_time	How often (in seconds) to generate new route addresses
log_handlers	Which log handlers (file, console) are in use
log_file	Log filename to use
log_level	Verbosity of logging output
users_location	Location of users auxiliary configuration file
ethers_location	Location of ethers auxiliary configuration file
profiles_location	Location of profiles auxiliary configuration file
crypt_type	Cryptographic algorithm to use
hash_type	Hashing function to use
regen_interval	How often (in seconds) to generate new session keys
regen_window	Window (in seconds) for session key exchange to take place
packets	Boolean value signifying packet manipulation is enabled
firewall	Boolean value signifying firewall manipulation is enabled
routes	Boolean value signifying route generation is enabled
flow_label	Boolean value signifying that flow labels are used on packets
Encrypt	Boolean value signifying that packet encryption is enabled
Urand	File handle to random number source

Table 3.1 - C MT6D Configuration Structure

The main thread then starts the other threads. The rehash thread is started first, followed by the ICMP thread. Then, one stream thread is started for each stream defined in the configuration file. Once the threads are started, the main thread waits to receive a termination signal from the operating system. Upon receipt of a signal, the main thread stops the other threads and terminates the program.

3.2.2 Rehash Thread

The main job of the rehash thread is to maintain each stream's routes. At startup,

however, the thread also performs some other tasks. The thread begins by adding a rule to ip6tables that directs all ICMPv6 traffic into the netfilter queue that is serviced by the ICMP thread. As mentioned in the Design Goals (section 3.1.1), the C MT6D program does not make external calls to other programs, such as the ip6tables executable. To add or remove a rule from ip6tables, the MT6D program uses functions made available through the ip6tc library. To help the reader understand, below is the equivalent ip6tables command that could be typed into a terminal. Table 3.2 gives an example ip6tables chain before starting MT6D, and Table 3.3 shows the effect of adding this rule. In this example, the ICMP thread is using queue number 1.

```
# ip6tables -I INPUT 1 -p ipv6-icmp -j NFQUEUE --queue-num 1
```

It is important to note that the rule is inserted with a priority of 1, causing the rule to appear at the top of the input chain. This arrangement gives it precedence over all other rules already loaded in the input chain. Later, when the stream threads start, they insert rules to direct stream (tunneled) traffic to the stream thread. The stream thread rules also have a priority of 1, which places them ahead of the ICMP rules. The end result, once the MT6D program has fully started, is that the initial MT6D rules are at the top of the input chain, with the stream rules first, followed by the ICMP rule. This ordering causes any packets that are part of a stream, including ICMP packets, to be directed to the relevant stream thread. All other ICMP packets, which are not part of a stream, are handled by the ICMP thread.

Chain INPUT (policy DROP)
prot: all; source: ::/0; destination: ::/0; state: RELATED,ESTABLISHED; target: ACCEPT

Table 3.2 - ip6tables Example Rules Before MT6D Startup

Chain INPUT (policy DROP)
prot: icmpv6; source: ::/0; destination: ::/0; target: NFQUEUE; num: 1
prot: all; source: ::/0; destination: ::/0; state: RELATED,ESTABLISHED; target: ACCEPT

Table 3.3 - ip6tables Rules After Inserting ICMP Rule

The next initialization task that the rehash thread performs is opening the RTNETLINK interface. The RTNETLINK interface is defined in the netlink library and is used to send messages to the Linux kernel to manipulate routing and addressing related structures. To open the interface, the rehash thread calls a helper function that interfaces with the netlink library to

open a NETLINK_ROUTE protocol socket. This socket is used later when address changes are needed.

The rehash thread then moves into the main loop of the thread. It starts by calculating the salt to be used during the current loop iteration. The salt is calculated using the current time, the rotation time specified in the program configuration, and an offset. The offset is not used (set to zero) on the very first iteration during startup. Equation 3.1describes the salt calculation [32].

$$salt = \frac{\left[current\_time - \left(current\_time \bmod 10\right)\right] + \left(offset \times rotation\_time\right)}{rotation\_time}$$

Equation 3.1 - Salt Calculation

The next step on the first loop iteration is to initialize each stream that is defined in the configuration. To initialize a stream, the rehash thread binds the stream destination address (that is, the true IPv6 address of the remote host that the local host communicates with behind MT6D) to the internal network interface, making it a local address on the MT6D device and causing the MT6D device to receive all packets destined to that address from the internal network. Next, the rehash thread creates a permanent entry in the neighbor cache for the stream source IPv6 and MAC addresses on the internal interface. This design removes the need for the MT6D device to use NDP packets to determine the link-layer address of the local host. Finally, the rehash thread creates an ip6tables rule that directs all traffic from the local host's true address to the remote host's true address into that stream's netfilter queue for processing by a stream thread. Shown below are the equivalent commands for binding the address, adding the cache entry, and inserting the ip6tables rule. Table 3.4 shows the rule added to the ip6tables chain.

```
# ip -6 addr add <stream destination>/64 dev <internal interface>
# ip -6 neigh add <stream source> lladdr <source MAC> nud permanent dev <internal interface>
# ip6tables -I INPUT 1 -s <stream source> -d <stream destination> -j NFQUEUE --queue-num
      <stream queue>
```

Chain INPUT (policy DROP)
prot: all; source: <stream source>; destination: <stream destination>; target: NFQUEUE; num: 2
prot: icmpv6; source: ::/0; destination: ::/0; target: NFQUEUE; num: 1
prot: all; source: ::/0; destination: ::/0; state: RELATED,ESTABLISHED; target: ACCEPT

Table 3.4 - ip6tables Rules After Inserting Stream Rule, Using Queue 2 as an Example

On all subsequent loop iterations, after calculating the salt, the rehash thread searches for expired routes to remove from each stream. It begins by garbage collecting expired routes from the previous iteration. For each previously expired route, the ip6tables rule for the route is removed and the route IPv6 address is unbound from its interface. Then the route structure is deleted. The rehash thread then searches for newly expired routes by the expiration time of each route in the route list. Once an expired route is found, the list is atomically separated and the expired list is saved to the expired pointer to be deleted during the next iteration.

The rehash thread is now ready to create new routes. After allocating a new route structure, the route source IPv6 address is created by hashing the session key, current salt, source MAC address, and true source IPv6 address. The route destination IPv6 address is created by hashing the session key, current salt, destination MAC address, and true destination IPv6 address. The route's expiration time is set by adding the current time to the route lifetime set in the configuration. The completed route structure is atomically added to the head of the route linked list for that stream, and, finally, the route source IPv6 address is bound to the network interface and a rule is added to ip6tables. Packets utilizing the new route can now be received, but packets are not sent using the route that was just created until the next iteration when it becomes the active route. The equivalent commands for binding the address and adding the ip6tables rule are shown below, and Table 3.5 shows the route rule added to the ip6tables chain.

```
# ip -6 addr add <route source>/64 dev <external interface>
# ip6tables -A INPUT -s <route destination> -d <route source> -j NFQUEUE --queue-num
      <stream queue>
```

Chain INPUT (policy DROP)
prot: all; source: <stream source>; destination: <stream destination>; target: NFQUEUE; num: 2
prot: icmpv6; source: ::/0; destination: ::/0; target: NFQUEUE; num: 1
prot: all; source: ::/0; destination: ::/0; state: RELATED,ESTABLISHED; target: ACCEPT
prot: all; source: <route source>; destination: <route destination>; target: NFQUEUE; num: 2

Table 3.5 - ip6tables Rules After Appending Route Rule

After repeating the preceding steps for each stream, the rehash thread has completed the current iteration of the main loop. If the current iteration was the initial iteration, the rehash

thread immediately executes the loop again. Otherwise, the rehash thread sleeps for the rotation time interval as specified in the configuration. The thread operation is summarized in Figure 3.5.

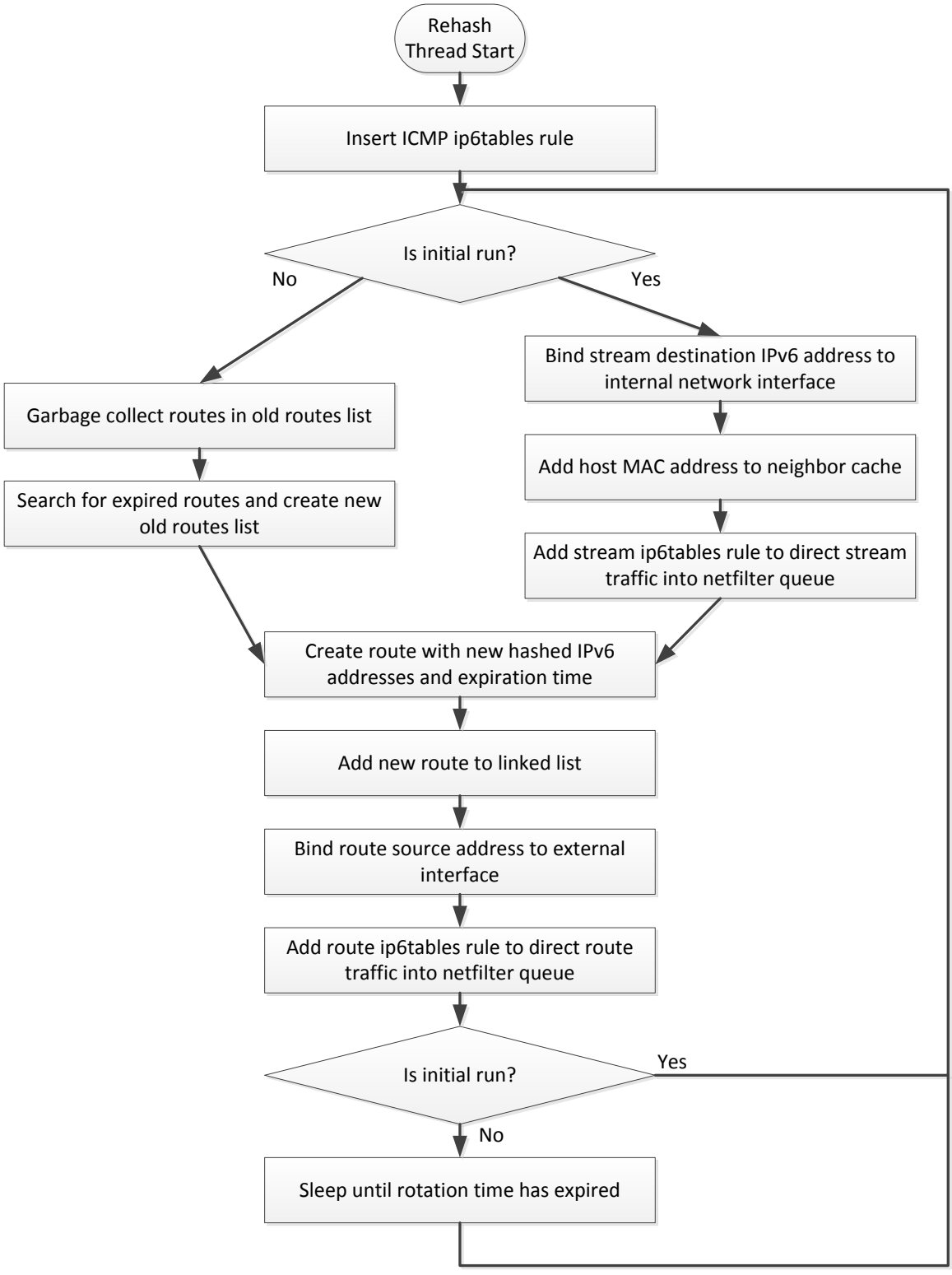

Figure 3.5 - C MT6D Rehash Thread Flow

3.2.3 ICMP Thread

The ICMP thread handles all MT6D related ICMPv6 packets, but does not include ICMPv6 packets between protected hosts. For example, if one protected host is pinging another, those packets would be handled by a stream thread, rather than the ICMP thread. The ICMP thread typically handles error packets from a hop along the path between the two MT6D devices.

The flow of the ICMP thread is illustrated in Figure 3.6. On startup, the first task of the ICMP thread is to register itself as the handler for the netfilter queue that ICMPv6 packets are directed into by the ip6tables rule that the rehash thread added. To register, the thread opens a handle to the netfilter queue library, binds itself as a queue handler, and opens a socket that is bound to the specific queue that is configured as the ICMP queue. The thread then waits for a

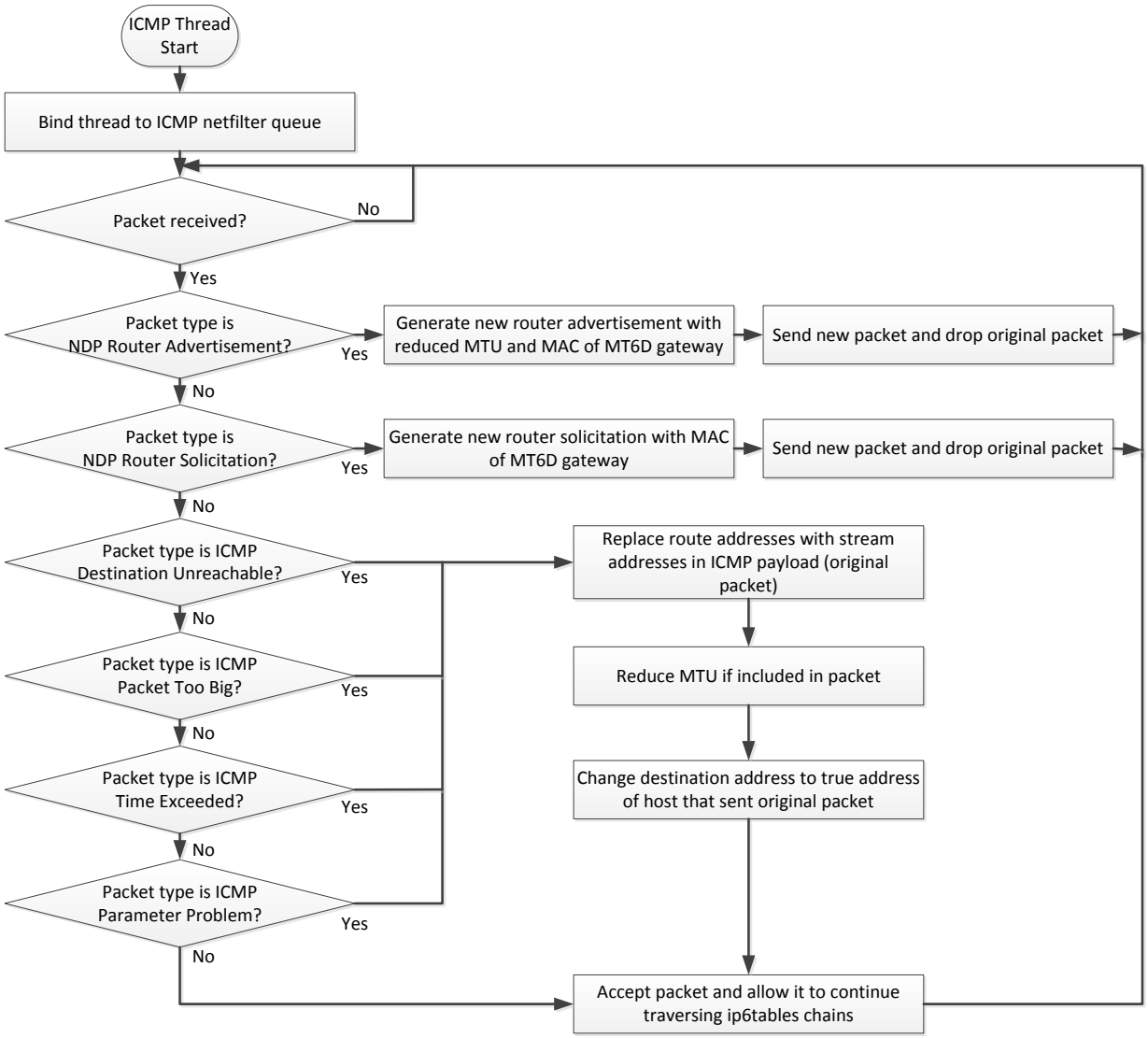

Figure 3.6 - C MT6D ICMP Thread Flow

packet to arrive in the queue by calling recv() on the queue's filehandle. When a packet is placed into the queue, the callback function of the thread that was registered as the queue handler is called, which is where the rest of the ICMP thread's work is done.

When the callback function is called to handle a packet, it first determines what the type code of the packet is, which dictates how the packet is handled. The first case considered is a NDP router advertisement. A router advertisement received from the external network must be modified before being forwarded to internal hosts. Specifically, the MTU must be reduced to accommodate the overhead imposed by MT6D, and the router's MAC address, if included, must be changed to the MAC address of the MT6D device so that internal hosts forward their packets through MT6D. To accomplish this process, the MT6D program generates a new router advertisement packet to replace the one received. The options and payload of the original packet are copied to the new packet, while checking the option types. If the MTU option is found, the value is changed to the MT6D value. If the source link-layer address option is found, the value is changed to the MAC address of the MT6D device's internal network interface. The rest of the options and payload are copied unchanged. After sending the new packet out into the internal network, the kernel is instructed to drop the original packet.

Another ICMPv6 packet type that receives special handling is the NDP router solicitation packet. Router solicitations must not pass because they typically contain the source link-layer address option with the MAC address of the sending host. Since the MAC address can uniquely identify a host on the network, which is what MT6D is designed to prevent, it is dropped and a new one is generated. Because router solicitations do not have to be received by the router from a specific host to trigger a router advertisement, MT6D simply generates a uniquely new one that is not copied from the original one at all. The ICMP thread generates a new router solicitation with the MAC address of the MT6D external network interface in the source link-layer address option and sends it to the external network.

The ICMP thread also handles error packets that are generated when a problem exists somewhere along the path between the two MT6D devices. Error types that are handled are "destination unreachable," "packet too big," "time exceeded," and "parameter problem." These errors must be delivered to the sending host so that the application that sent the original packet can take corrective action. However, the error packet contains information from outside the MT6D tunnel that the end host should not receive, so it must be modified first. The ICMP thread

first extracts the original packet from the tunnel packet (which is the packet that caused the error) in the ICMPv6 payload and changes the IPv6 addresses from the hashed route addresses to the true addresses that the host would have used when sending the packet. If the error type is "packet too big," the ICMP thread also reduces the MTU contained in the packet by the size of the MT6D overhead. The destination address on the packet is changed from the route address to the true address of the sending host, and the packet is sent via the internal interface. When the sending host receives the packet, no MT6D tunnel information or route addresses are present in the packet.

These ICMP thread actions hide the presence of MT6D and the tunneling from the hosts behind MT6D, but still allow the hosts to receive notification of transmission errors. After receiving the error notice, the host's network stack or application can fix the problem and retransmit or inform the user. The ICMP thread allows all other types of ICMPv6 packets to continue through ip6tables rule processing and eventually be delivered as they are either not relevant to MT6D or do not require modification.

3.2.4 Stream Thread

The stream thread is responsible for handling all packets that are part of the MT6D tunnel. Packets that are received on the internal interface are tunneled inside an MT6D packet for transmission over the external interface to the remote MT6D device. MT6D packets received on the external interface are extracted from the tunneled packet and prepared for transmission over the internal interface to the protected host. The MT6D program starts one stream thread and allocates one netfilter queue for each stream that is defined in the configuration. Each stream thread is bound to a specific queue. Due to the number of modifications to packets that is necessary to tunnel packets in MT6D, all packets received in the stream queues are dropped and new ones are eventually transmitted, rather than attempting to modify the existing packet and allowing it to pass.

The flow of the stream thread is illustrated in Figure 3.7. At startup, the stream thread performs several size calculations and memory allocations that are used later in the callback function when a packet is received. This action is completed to improve performance – the structures are all reusable and allocating them once at startup, rather than continually allocating and freeing for each packet received, is more efficient. Two sockets are opened first – one raw

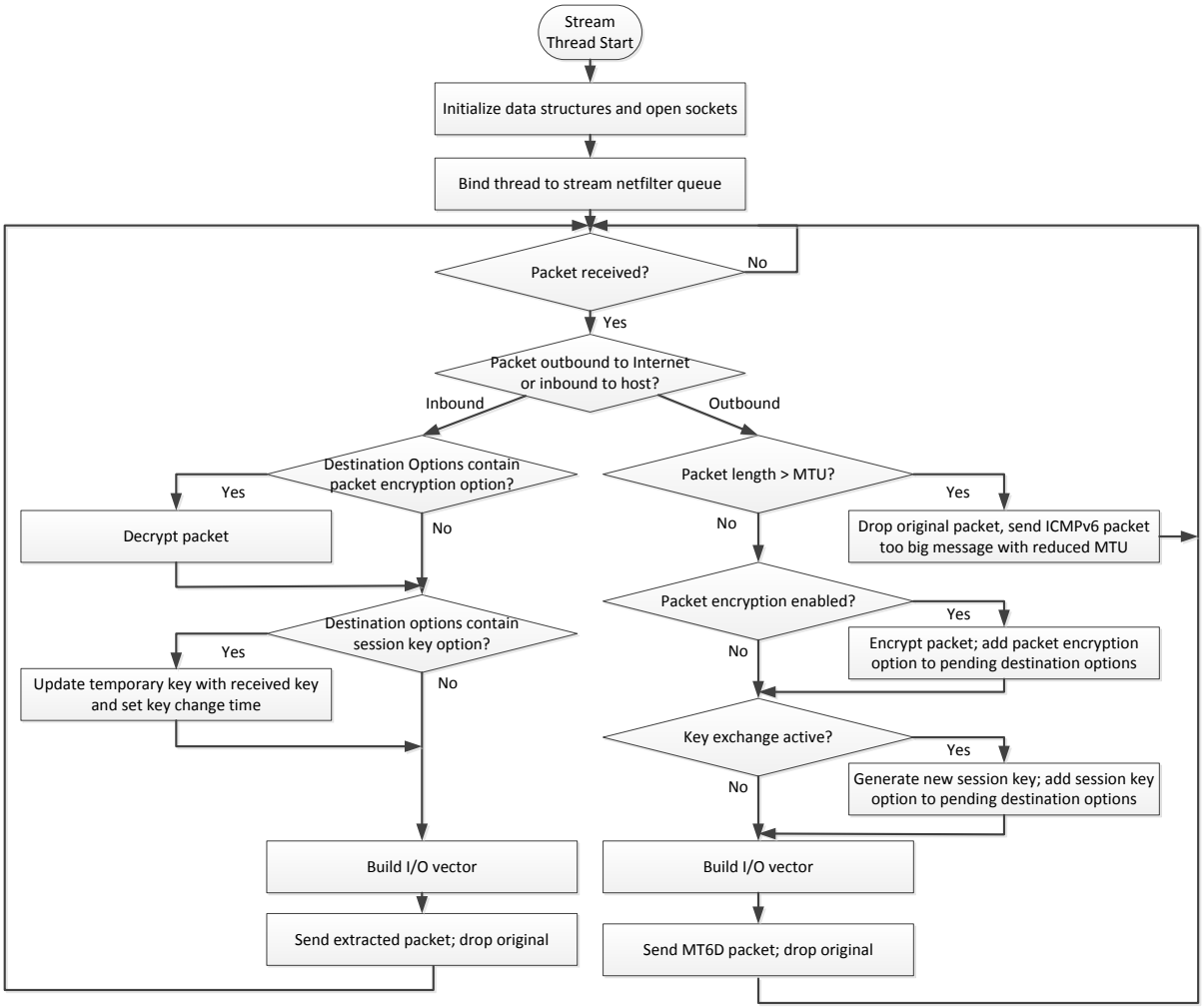

Figure 3.7 - C MT6D Stream Thread Flow

socket for transmitting packets on the internal interface and one UDP socket for transmitting packets on the external interface. The OpenSSL context used for encryption is initialized and set up for encrypting and decrypting. This action is necessary even if packet encryption is not enabled, as session key exchanges also use encryption. Buffer sizes needed to hold destination options with and without a session key are pre-computed. Finally, the input and output (I/O) vectors and message headers used to pass additional options to the transmission function (sendmsg()) are initialized and preset with default options. These structures are reused throughout the program by changing pointer addresses. Once these initialization steps are complete, the stream thread opens a handle to the netfilter queue library and binds its callback function to its assigned queue in much the same way as the rehash thread.

The stream thread callback function begins by determining the direction of the packet received. Because packets are directed to the stream thread queue by IPv6 address matching ip6tables rules, the stream thread need not verify that the packet is part of a valid MT6D stream or which stream it is a part of. To determine the packet direction, the stream thread compares the packet source and destination address with the true stream addresses. If they match, the packet is coming from the internal network and needs to be tunneled. If the addresses do not match, then the packet has hashed route addresses and is coming from the external network and, therefore, needs to be extracted.

For packets that are outbound (coming from the internal network), the stream thread checks the size of the packet to verify that it does not exceed the MT6D MTU. If the packet is too big, the stream thread generates a new ICMPv6 packet "too big message," sends it to the source host, and then drops the original packet. Otherwise, the stream thread prepares to tunnel the packet inside an MT6D packet. First, the stream thread checks the state of the session key. The session key state, indicating the progress of a key exchange, determines the UDP port number the packet uses to indicate the key state to the receiving MT6D device. If packet encryption is enabled, the packet is encrypted in two parts. The first part consists of the IPv6 header before the IPv6 source and destination addresses, and the second part is the remainder of the packet after the addresses. Excluding the IPv6 addresses is possible because the remote MT6D device knows what true addresses need to be reinserted into the packet; in addition, not transmitting the addresses reduces the MT6D overhead.

Next, the destination options are built. MT6D uses IPv6 destination options to communicate encryption settings and session key exchanges to the remote MT6D device. If packet encryption is not enabled and a key exchange is not taking place, no destination options are sent with the packet. If packet encryption is enabled, a destination option indicating the presence of encryption and the encryption algorithm used is generated. If a session key exchange is in progress, a destination option containing the encrypted new session key and key change time is generated. One or both of these destination options may be sent with a packet as required.

The stream thread now prepares the data structures needed for transmission, which were already allocated during startup. The C MT6D implementation uses the sendmsg() system call to transmit packets which, while more complex, allows the transmission of ancillary data (i.e., IPv6

destination options) and the use of vectored I/O [33]. When packet encryption is enabled, the two parts of the packet (the header before the addresses and the payload after) are combined in the encryption operation into one buffer for transmission. The I/O vector has only one entry, which will point to the buffer containing the encrypted packet. When packet encryption is not enabled, the entire packet is located in one buffer, but the IPv6 addresses in the header should not be transmitted. In this case, the I/O vector will have two entries: one pointing to the beginning of the packet but only 8 bytes long (40 byte IPv6 header minus two 16 byte IPv6 addresses) and one pointing to the byte after the end of the IPv6 header.

Control message structure is built next. If any destination options were generated, the structures describing those are linked into the control message structure, along with a packet information structure that indicates which network interface and source address to use in sending the packet. The control message structure and I/O vector structure are linked into a message header structure and passed, along with the previously opened UDP socket, to the sendmsg() function, which sends the packet to the external network.

For packets that are inbound (coming from the external network), the stream thread iterates through each header in the packet, recording the location of a destination options header, if it exists, and continuing until it finds a UDP header or reaches the end of the packet. Since MT6D uses UDP as the transport layer protocol, a packet without a UDP header is classified as a malformed packet and is dropped. Once the UDP header is located, the destination port is read to determine if a key exchange state change is necessary.

If the initial header search found a destination options header, it is processed next. If packet encryption is enabled, a destination option should indicate this situation, as well as identify which encryption algorithm was used; then, the packet is decrypted. The destination options may also contain a session key exchange. If the internal session key exchange state indicates that a new key should be received, the destination option is processed and the included key is decrypted. The decrypted key will be stored in temporary memory until the key exchange is complete and the stream is updated to use the new key.

Once destination options processing is complete, the packet is ready is sent to the destination host. The IPv6 addresses are still missing from the packet, as they are not included in the tunneled packet's header when it is transmitted across the external network. Rather than perform the additional steps of copying the true stream addresses into the proper position in the

packet, the I/O vector is used to assemble the complete packet. In this case, the vector has four entries. The first entry points to the 8 byte IPv6 header portion of the received packet. The second and third entries point to the memory location in the stream structure where the true IPv6 addresses are stored. The fourth entry points to the remainder of the received packet. See Figure 3.8 for an example of the I/O vector in this case. As with tunneling a packet, the control message structure (describing which source address to use) and I/O vector structure are linked into a message header structure and passed, along with the previously opened raw socket, to the sendmsg() function, which sends the packet to the internal network.

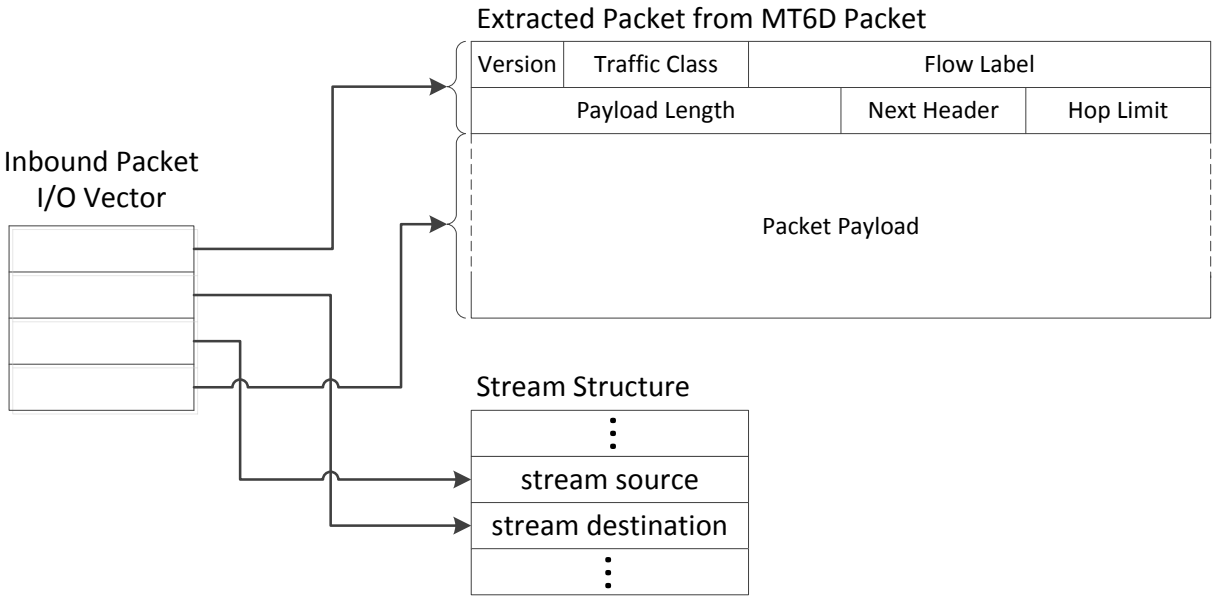

Figure 3.8 - Scatter/Gather I/O Vector

3.3 Stream Flow

To illustrate the operation of C MT6D, this section describes a communication session between two hosts behind MT6D. The first host opens a Hypertext Transfer Protocol (HTTP) connection to the second host and transfers enough data that the session continues through one address rotation. The details of the two hosts are shown below.

Host A
Hostname: mt6d-host-a
IPv6 Address: 2001:468:c80:c111::1
MAC Address: 00:ab:cd:11:22:01

Host B
Hostname: mt6d-host-b
IPv6 Address: 2001:468:c80:c111::2
MAC Address: 00:ab:cd:11:22:02

Each host is behind an MT6D device that separates the hosts from the Internet, called an MT6D gateway. The gateway details are shown below.

Gateway A
Hostname: mt6d-gw-a
Internal Network: eth1 connected to Host A
Internal MAC Address: 00:ab:cd:11:33:01
External Network: eth0 connected to Internet
External MAC Address: 00:ab:cd:11:44:01

Gateway B
Hostname: mt6d-gw-b
Internal Network: eth1 connected to Host B
Internal MAC Address: 00:ab:cd:11:33:02
External Network: eth0 connected to Internet
External MAC Address: 00:ab:cd:11:44:02

The gateways communicate with each other over the Internet by using a multi-hop, possibly changing route. The network layout is illustrated in Figure 3.9.

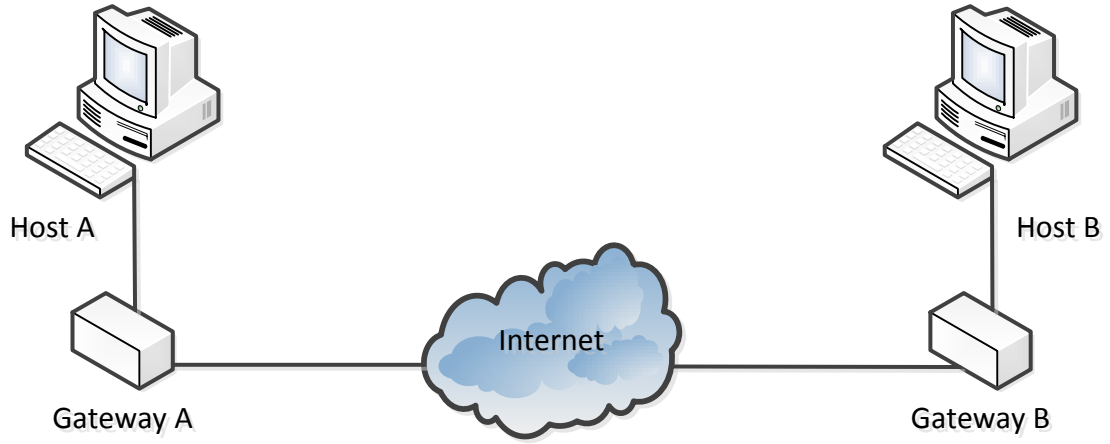

Figure 3.9 - MT6D Example Network Layout

This setup is defined in the relevant configuration files of both MT6D gateways, and a session key is generated for the stream. The sequence of steps taken on Gateway A during MT6D program startup is described below. The steps on Gateway B are the same, but with the addresses swapped.

First, the rehash thread starts and inserts the ICMP rule into the top of the ip6tables input chain. This insertion is done programmatically, but the equivalent command that could be executed from a command prompt is

```
# ip6tables -I INPUT 1 -p ipv6-icmp -j NFQUEUE --queue-num 1
```

Next, the rehash thread will enter its main loop and begin setting up each stream defined in the configuration. As this is the initial execution of the loop, the rehash thread will bind Host

B's IPv6 address to the internal interface and add Host A's MAC address to the neighbor cache. It will also add the ip6tables rule that directs stream traffic into the netfilter queue that the stream thread is bound to. The equivalent commands are listed below.

```
# ip -6 addr add 2001:468:c80:c111::2/64 dev eth1
# ip -6 neigh add 2001:468:c80:c111::1 lladdr 00:ab:cd:11:22:01 nud permanent dev eth1
# ip6tables -I INPUT 1 -s 2001:468:c80:c111::1 -d 2001:468:c80:c111::2 -j NFQUEUE --queue-
        num 2
```

The rehash thread now creates a route with hashed addresses to use when sending packets to the external network. As this is the initial run, two routes are created. The exact route addresses created are dependent on the true IPv6 address, current time, session key, and MAC address. The route source addresses are bound to the external interface and a new ip6tables rule is appended to the input chain. The equivalent commands are listed below.

```
# ip -6 addr add 2001:468:c80:c111:886b:14a9:cc13:8991/64 dev eth0
# ip6tables -A INPUT -s 2001:468:c80:c111:f93b:abcc:ac79:2613 -d
        2001:468:c80:c111:886b:14a9:cc13:8991 -j NFQUEUE --queue-num 2
# ip -6 addr add 2001:468:c80:c111:9988:1463:cccc:1b62/64 dev eth0
# ip6tables -A INPUT -s 2001:468:c80:c111:6197:abbf:ac16:9898 -d
        2001:468:c80:c111:9988:1463:cccc:1b62 -j NFQUEUE --queue-num 2
```

The work of the rehash thread is now complete. It sleeps until the rotation time has expired, when it generates the next new set of addresses. In the meantime, the ICMP thread has started and is now bound to the ICMP netfilter queue (queue number 1) and is waiting for packets. The stream thread has also started, and, after initializing data structures and binding to the stream queue (queue number 2), it is also waiting for packets. At this point, the MT6D gateways are ready to begin forwarding packets.

Host A now prepares to open a transmission control protocol (TCP) connection to carry HTTP traffic to Host B. First, Host A sends an NDP neighbor solicitation packet to the all-nodes multicast group with Host B's IPv6 address (2001:468:c80:c111::2) as the target address. Because Host B's IPv6 address is bound to the internal interface of Gateway A, the networking stack on Gateway A, independent of MT6D, responds with an NDP neighbor advertisement, with the target IPv6 address contained in the neighbor solicitation and the target link-layer address set to the MAC address of Gateway A's internal interface. Host A can now begin IPv6 communications with Host B.

Host A begins communicating with Host B as the first packet is the start of the TCP three-way handshake. This packet has a source address of 2001:468:c80:c111::1 and a destination address of 2001:468:c80:c111::2 and is sent by Host A to Gateway A and placed in netfilter queue 2 by the ip6tables rules on Gateway A. The stream thread tunnels this packet inside an MT6D packet, removing the original IPv6 addresses from the inner packet in the process. Using the active route, the MT6D packet sent over the external interface to Gateway B uses a source address of 2001:468:c80:c111:886b:14a9:cc13:8991 and a destination address of 2001:468:c80:c111:f93b:abcc:ac79:2613.

After travelling through the Internet, the packet is received by Gateway B and placed in netfilter queue 2 by the ip6tables rules on Gateway B. The tunneled packet is extracted from the MT6D packet and sent via the internal interface to Host B with a source address of 2001:468:c80:c111::1 and a destination address of 2001:468:c80:c111::2. This packet is received by Host B, and, as far as it knows, the packet came directly from Host A. Host B responds with the second packet in the TCP handshake, and the packet is transmitted in the same manner back to Host A.

Once the TCP connection is established, Host A establishes an HTTP connection to Host B and requests a large file, which Host B begins transmitting to Host A. While the data are being transmitted, the rotation time interval expires and the rehash thread is awakened to create new routes. The rehash thread starts by searching for expired routes, but, as this is only the third iteration through the main loop, the rehash thread finds that none of the routes have expired yet. Next, the rehash thread generates a new route for the stream, binds the new address, and inserts the ip6tables rule. The equivalent commands are listed below.

```
# ip -6 addr add 2001:468:c80:c111:6611:e4b9:b9d9:1647/64 dev eth0
# ip6tables -A INPUT -s 2001:468:c80:c111:0011:b600:22aa:1942 -d
        2001:468:c80:c111:6611:e4b9:b9d9:1647 -j NFQUEUE --queue-num 2
```

Once the routes are generated, the rehash thread updates the route pointers in the stream structure. First, the next pointer in the new route is set to point to the current head of the routes list. Next, the active route pointer is atomically changed to point to the current head of the routes list. Finally, the routes list head is atomically changed to point to the new route. Once the active route pointer is changed, the packets in the HTTP data stream immediately begin using a different set of addresses. In this example, the source address changes from

2001:468:c80:c111:886b:14a9:cc13:8991 to 2001:468:c80:c111:9988:1463:cccc:1b62 and the destination address changes from 2001:468:c80:c111:f93b:abcc:ac79:2613 to 2001:468:c80:c111:6197:abbf:ac16:9898. This new set of addresses is the second newest set (which is now the active set) and was generated by the second execution of the rehash thread loop during thread startup.

The route address change is invisible to Host A and Host B and does not interfere with the TCP connection in any way. The source and destination addresses used by Host A and Host B do not change, and so the endpoints of the TCP connection do not change. Because the second most recent set of route addresses are used as the active route, both MT6D gateway devices have had at least one rotation time interval to bind the addresses to their network interfaces and perform duplicate address detection and any other steps the operating system may perform when assigning new addresses. This result should mean that the addresses are immediately usable and no packets will be dropped when MT6D switches to those addresses. If some packets are dropped, the MT6D transport protocol (UDP) does not notice, but the tunneled transport protocol of the communicating hosts (TCP) detects the lost packets and simply triggers a retransmission, as it is designed to do.

Chapter 4: Testing and Analysis

To evaluate the overall performance of C MT6D, including the effectiveness of the specific design choices described in the previous chapter (Chapter 3), a suite of tests are designed. The tests allow for comparisons between C MT6D and Python MT6D, as well as between C MT6D and no MT6D. In addition to network metrics (such as bandwidth and latency), several host metrics are monitored to determine the effect of running MT6D on a specific host. The results of the tests evaluated to see if the C version is able to offer substantial efficiencies and if some of the specific points of the C version's design make a difference.

In all of the tests, the basic topology used, as shown in Figure 4.1, consists of two devices running the MT6D software in gateway mode and one host device behind each MT6D device. Host A is where the tests are initiated and their results recorded, with the target of the tests being Host B. When profiling is used, the software on Gateway A is profiled. The network connection between the two MT6D gateway devices is changed to create different test scenarios, as described later in this chapter.

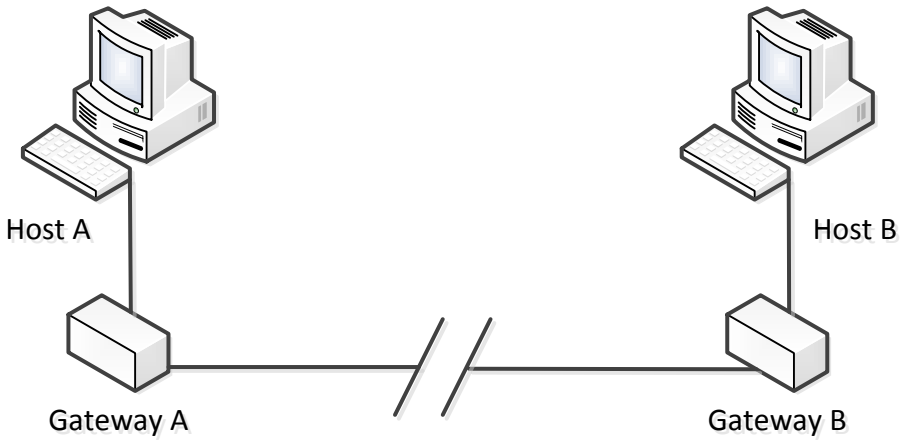

Figure 4.1 - Basic Testing Topology

Each type of test is run 10 times for each MT6D scenario – first with no MT6D running, second with Python MT6D, and third with C MT6D. In the first case where no MT6D is used, the two gateway devices are configured as simple network bridges to pass all traffic from one side to the other without modification. This setup allows for a performance baseline to be

established before MT6D is introduced. The MT6D settings are kept the same throughout the tests and between the Python and C versions – packet encryption is disabled, the SHA-256 hashing function is used, the session key is regenerated every 86400 ± 30 seconds, and the MT6D addresses are mutated every 10 seconds.

The hardware used in the tests is the GuruPlug Server [34], manufactured by GlobalScale Technologies. The GuruPlug Server is used for the two hosts and the GuruPlug Server Plus (which has two Ethernet ports) is used for the gateway devices. The GuruPlug Server is a small, wall plug sized, embedded system containing a Marvell Kirkwood 6281 (ARM) central processing unit (CPU) running at 1.2 GHz, 512 MB double data rate (DDR2) 800 MHz random access memory (RAM), two universal serial bus (USB) 2.0 ports, and one or two gigabit Ethernet ports. It also contains NAND flash memory as its primary persistent storage, but these tests use a root filesystem contained on a USB flash drive and the internal flash memory is not used. The devices run the Angstrom Distribution, version 2011.09, with Linux kernel 2.6.37.6 and are built using the OpenEmbedded build framework [35]. The actual installed software is a minimal install, with only the packages necessary to run MT6D, the tests, and the instrumentation installed. For further information about the build environment, see Appendix B.

All tests are conducted on the Virginia Tech production IPv6 network. SLAAC is used to assign globally unique IPv6 addresses to each network device, and the network is connected to the IPv6 Internet and carries live traffic.

4.1 Network Topologies

During testing, three network topologies (referred to as switched, routed, and tunneled) are used to separate the two MT6D gateway nodes. The topologies provide varying levels of exposure to external network traffic and intermediate hops and show varying levels of performance and compatibility with MT6D.

The switched topology is the simplest, where the two MT6D gateways are connected to each other through a single switch, as shown in Figure 4.2.

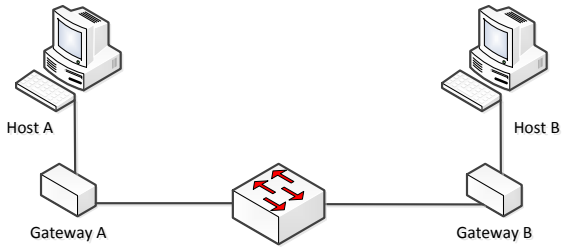

Figure 4.2 - Switched Testing Topology

In this topology, the MT6D traffic is not exposed to any other network traffic and contains no layer 3 hops in the path. The tests that use this topology are intended to show the best possible performance of MT6D on these hosts. The compatibility of MT6D with routers and other network equipment is not tested here.

Routed topology builds on the switched topology and is slightly more complex, as shown in Figure 4.3.

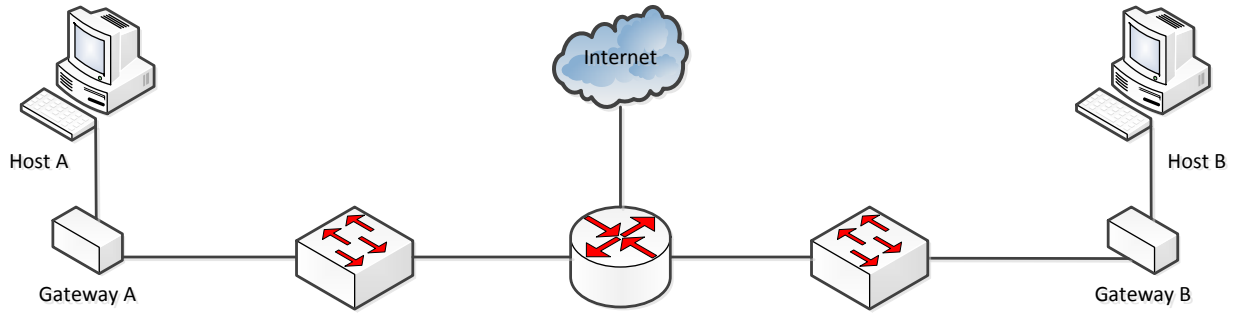

Figure 4.3 - Routed Testing Topology

In this topology, MT6D traffic that leaves one of the gateways will pass through at least one switch, go through one of the main routers on Virginia Tech's IPv6 network, and pass through at least one more switch before reaching the other gateway. These switches and the router are also carrying production traffic from many other systems, and the Internet is accessible through the router.

The purpose of the routed topology is to expose the MT6D traffic to other, unrelated traffic that exists in the switches and router. This traffic can introduce unexpected latencies and packet loss, which MT6D must be able to handle, while not impacting the overall performance significantly. Also, the presence of the router tests the correctness of the layer 3 headers of the MT6D traffic and also tests the MT6D gateway's ability to handle router advertisements.

Tunneled topology builds on routed topology and is the most complex. As shown in Figure 4.4, one of the MT6D gateway hosts remains connected to Virginia Tech's IPv6 network, while the other gateway host is directed through a 6in4 tunnel provided by Hurricane Electric [36]. The red parts of the path in the figure are where IPv6 packets are tunneled in IPv4 packets. The entire red path appears as a single hop to IPv6.

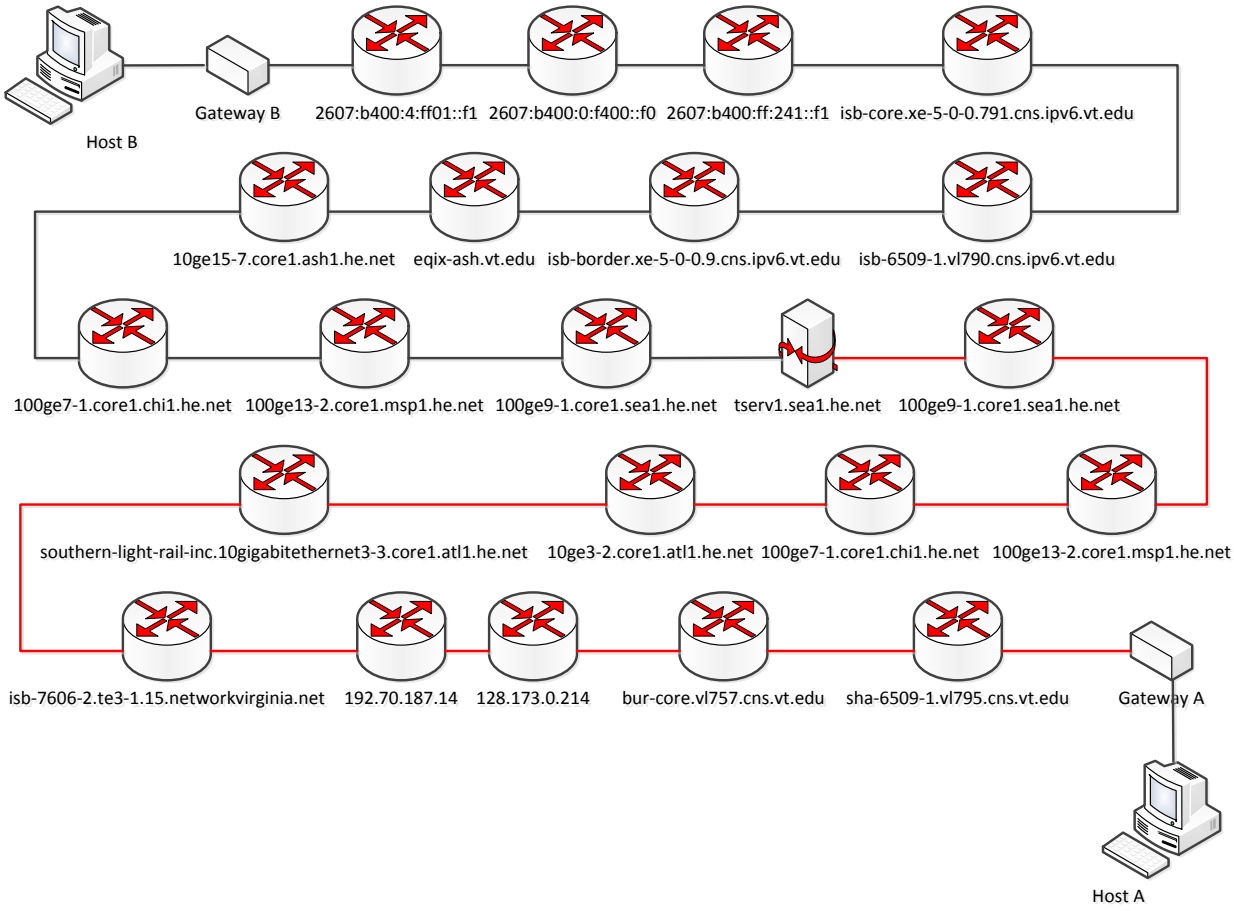

Figure 4.4 - Tunneled Testing Topology

During tunnel setup, Hurricane Electric allows a choice of which geographically diverse tunnel endpoint to use. For these tests, the tunnel endpoint in Seattle, Washington, was chosen. This choice requires packets to traverse the United States twice to travel from Gateway A to Gateway B, providing a diverse path with many other types of unrelated traffic along the way. The use of a 6in4 tunnel also requires the MT6D software to handle a reduced MTU, which is necessary to accommodate the 6in4 tunnel overhead. The network metrics being measured are

expected to show less performance on this network topology due to the longer round trip distance and quantity of unrelated traffic.

4.2 Traffic Types

Three different types of traffic (ICMP, TCP, and UDP) are tested, which represent realistic traffic patterns that a production MT6D setup might be expected to handle. Each type is also used to collect a specific network performance metric, as described in the following sections.

4.2.1 Internet Control Message Protocol (ICMP)

The ping command is used to generate ICMP traffic (echo requests and replies). Three different tests use the ping command – standard, flood (10k), and flood (50k). The standard ping test sends 1,000 64 byte packets with a one-second delay between packets. This test is intended to show the latency and packet loss of the path under a very light load. The latency during this test is expected to be very low and very constant, especially in the switched topology, so that any variations are caused by MT6D. Packet loss should stay at zero percent. The International Telecommunication Union (ITU), in its recommendation G.114, suggests that one-way latencies should be less than 150 milliseconds for acceptable transmission of real-time data, such as Voice over Internet Protocol (VoIP) [37]. As communication is a likely application where a user's privacy should be protected, MT6D should be able to maintain a latency that is low enough to permit the transfer of real-time data.

The second two tests send floods of 64 byte packets (10,000 or 50,000 total packets, respectively) from one host to the other. In the flood tests, packets are sent as fast as they come back, or 100 times per second, whichever is greater. This test is intended to show the latency and packet loss of the path during periods of heavy traffic, although the traffic size is small. Ideally, the latency should remain unchanged from the standard ping test and packet loss should stay at zero percent, but if delays are introduced by MT6D, for example during address rotations, they will be detected during this test. It is expected that the C version will have a lower packet loss and less deviation in the latency than the Python version.

4.2.2 Transmission Control Protocol (TCP)

In order to test TCP connections, the wget command is used to download files of various sizes over a HTTP session. Host A is the client where wget is run, and Host B is the server where the Apache HTTPD web server software is running. Pre-generated files of random data in sizes 500 kB, 1 MB, 10 MB, 50 MB, and 500 MB are available for download. The TCP tests download each file size an equal number of times. The various file sizes are used to generate variable timed HTTP sessions so that both short-lived and long-lived TCP connections can be tested. This test focuses on the bandwidth of the connection observed while downloading the complete file. Ideally, using MT6D should decrease the available bandwidth as little as possible over the base scenario (no MT6D). The C version is expected to have significantly higher bandwidth than the Python version.

4.2.3 User Datagram Protocol (UDP)

The Iperf tool is used to test packet latency variation (jitter), using specially generated UDP packets sent from Iperf running in server mode on Host B to the client on Host A. Three versions of the test are done, transferring 1 MB, 10 MB, or 100 MB of data between the two hosts. Having a low jitter is also crucial for real-time data transfer; the ideal is packet loss remaining at zero percent and jitter minimal. The Python version is expected to have a higher jitter than the C version, due to its method of rotation of IPv6 addresses.

4.3 Host Performance

In addition to the network tests discussed above, additional tests measure certain metrics on the host running MT6D (Gateway A and Gateway B in the first figure). Three metrics are measured – CPU usage, memory usage, and kernel/user time. MT6D in gateway mode was designed to run on a device between two communicating hosts, such as a router, which is typically a small, embedded system with limited resources. Therefore, MT6D must be designed to minimize the load on the host's CPU and the amount of memory it consumes.

The CPU usage of MT6D is collected using the ps program, just before killing the MT6D executable after a test run. This measurement gives the average CPU usage during the life of the MT6D executable, which is restarted for each test run. The ps program calculates the CPU usage using Equation 4.1and the variables shown below [38]. As used here, "jiffy" refers to the time

between two interrupts of the system timer. On the GuruPlugs used for these tests, the system timer operates at 100 Hz.

- UTIME: Amount of time (in jiffies) MT6D is scheduled in user mode
- STIME: Amount of time (in jiffies) MT6D is scheduled in kernel mode
- DBOOT: Amount of time (in seconds) since system boot
- PTIME: Process start time (in jiffies), relative to system boot
- HZ: Timer interrupt frequency (in hertz)

$$\text{CPU Usage} = \frac{\left[\left(UTIME + STIME\right) * 100\right] / HZ}{DBOOT - \left(PTIME / HZ\right)}$$

Equation 4.1 - ps CPU Usage Calculation

During the design of C MT6D, CPU usage was not specifically targeted for improvement beyond the changes discussed in Chapter 3, but it was assumed that simply writing the program in C would reduce its CPU usage over the Python version. However, memory usage is addressed in the C version. The memory footprint is kept small, memory is freed as soon as possible, temporary objects are avoided, and the frequency of allocations is reduced by pre-allocating and reusing certain objects. Due to these considerations, the C version is expected to use significantly less memory than the Python version.

The memory usage of MT6D is recorded in two ways. The first is by using the heap profiler from Gperftools, which is designed and used by Google [39]. The heap profiler is loaded at MT6D runtime and creates periodic profiles that are analyzed using a separate program after test run completion. The second method makes use of the Linux kernel's page monitor functionality (and the /proc/[pid]/smaps file), which reports the memory consumption for each memory mapping of a process [40]. In both cases, only the private memory (memory that is not marked as shared with other processes) is recorded.

The third metric, kernel/user time, refers to the amount of time the CPU is executing code in kernel space versus user space. One goal of the C version is to move as much work as possible from the MT6D program to the Linux kernel. One example of this move is neighbor discovery packets – the Python version generates these packets itself, but the C version binds the remote address to the local interface so that the network stack will generate these packets

automatically. Also, it is assumed that the Python version, which is dependent on many other libraries (including the Python library itself), will spend a significant amount of time executing code in these libraries. The C version is designed to depend minimally on other libraries (other than the C library, but even this use should be minimal).

The kernel/user time metric is collected using the performance counters that are built into the Linux kernel [41]. This structure is an event-based sampling system, with a sample recorded whenever the performance counter in use overflows. Sampling data are analyzed by a separate program after the test run is complete. The analysis determines what symbol is being executed at the time the sample is taken and the location of the symbol (MT6D executable, library, or kernel). Summing and grouping by source give a measurement of how much time is spent in kernel code versus user code.

4.4 Summary

Tests described in this chapter are designed to stress various aspects of the MT6D program and determine if the C version is improved in specific areas over the Python version. They are also designed to show how an implementation of a MTD uses the resources on a host system and where attention is needed so that an MTD can be implemented on a resource constrained system. The three network topologies (switched, routed, and tunneled) are combined with the tests for each traffic type (ICMP, TCP, and UDP) and host performance metric (CPU usage, memory usage, and kernel/user time) to show the overall performance of MT6D under realistic network conditions. Table 4.1 lists each type of test that is run, and Table 4.2 lists the combinations of MT6D, topology, and host metric used.

Traffic Test Names	
ping_std_1k	wget_50M
ping_flood_10k	wget_100M
ping_flood_50k	iperf_1M
wget_500k	iperf_10M
wget_1M	iperf_100M
wget_10M	

Table 4.1 - Traffic Test Names

Network Topology	Host Metric			
	None	CPU Usage	Memory Usage	Kernel/ User Time
Switched				
Routed				
Tunneled				

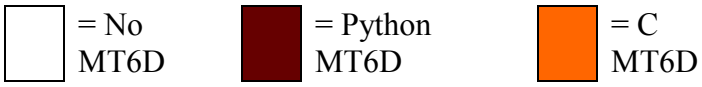

Table 4.2 - Test Topology/Metric/MT6D Combinations

Chapter 5: Analysis of Results

Tests described in the previous chapter were run over an extended time interval in each scenario, generating over 140 GB of data. Data from the 10 iterations of each test were averaged together, and the results are presented in the following sections.

5.1 Switched Topology

As described in the previous chapter, the first series of tests were run under the switched network topology, where the two gateway devices are separated by a single network switch. Tests run under this topology are expected to show the best performance since the path between the two hosts is short and has minimal unrelated traffic.

5.1.1 No MT6D

The first set of tests run under the switched topology was run without using MT6D. The two gateway devices were configured to simply pass traffic from one interface to the other as a network bridge. These tests will be used to establish a baseline for performance that all of the following tests can be compared with.

Figure 5.1 shows the results of the ICMP tests. This figure plots the minimum, average, and maximum latencies, as well as packet loss observed during the three tests. The latency vertical axis uses a logarithmic scale to allow the three latency values easier visibility. As expected, the results of this test show a stable, low latency network between Host A and Host B. No packets were lost during any of these tests, and the average latency is less than 1 millisecond. Results indicate that much larger maximum times occurred at the beginning of the tests when a neighbor discovery packet was sent to determine the MAC address of the hosts.

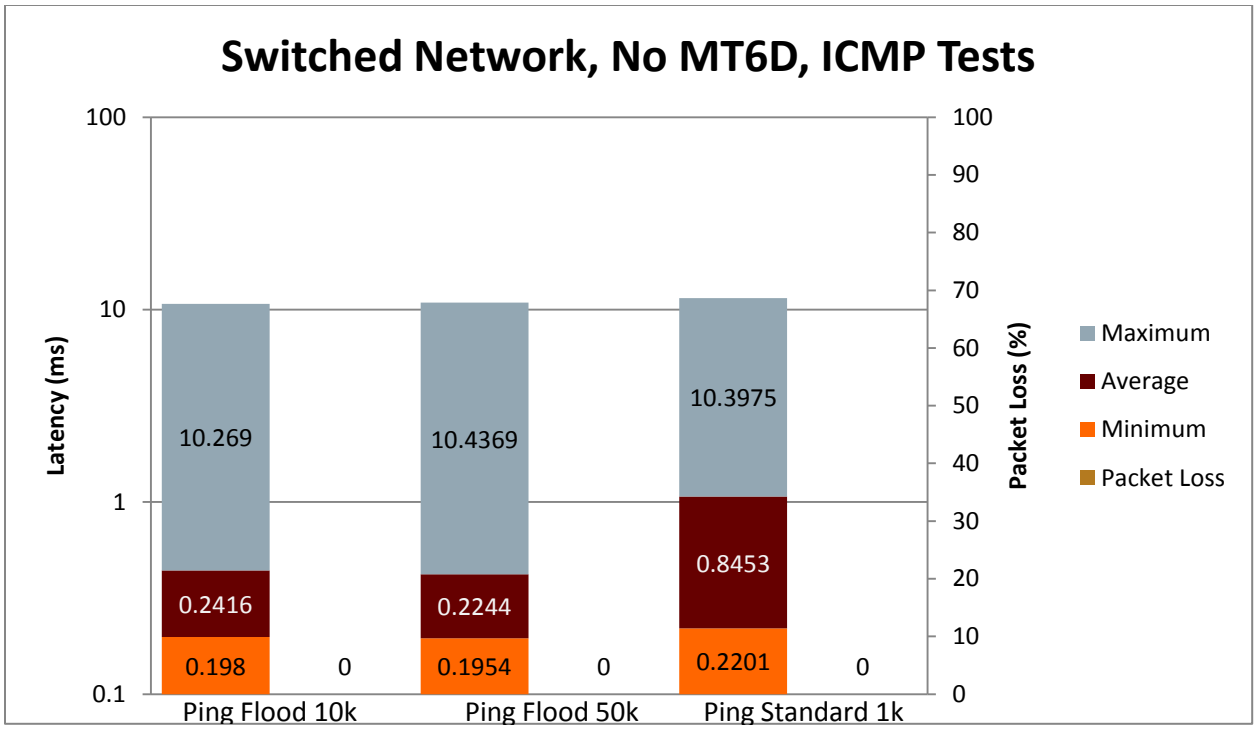

Figure 5.1 - Results of Switched Network, No MT6D, ICMP Tests

Figure 5.2 shows the results of the TCP tests. Each bar on the graph represents a different data file size that is being fetched via the HTTP protocol. As expected, the throughput is fairly consistent across all the tests, except the largest one. However, the most important result from these tests is the overall average throughput, which is 19.584 megabytes per second. Translated into bits, the speed is 156.672 megabits per second, which is significantly less than anticipated. All four GuruPlug devices used in this test for the hosts and gateways have gigabit network interfaces, and the switch connecting the gateways is a gigabit switch with more than sufficient switching capacity, leading to an anticipated throughput close to 1 gigabit per second. Unfortunately, the GuruPlug devices are simply not capable of passing large amounts of data any faster. These results indicate that the baseline throughput, which will be compared with future results, is only 19.584 megabytes per second.

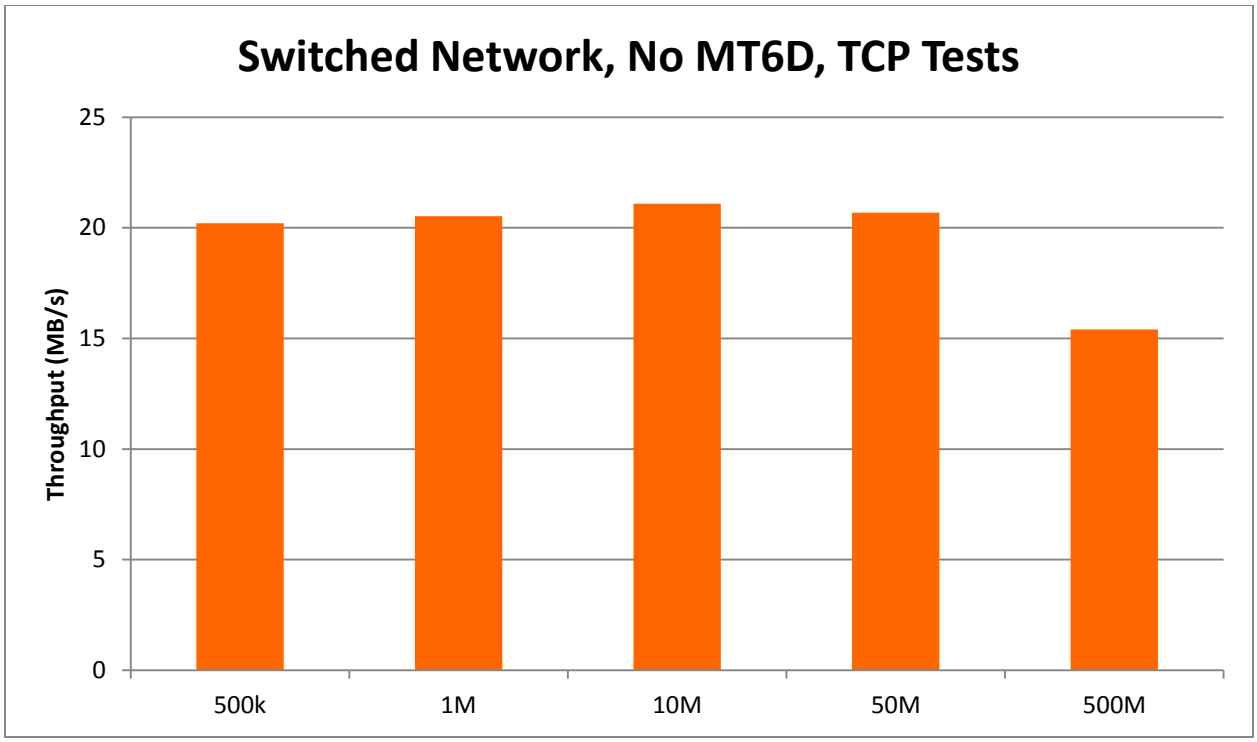

Figure 5.2 - Results of Switched Network, No MT6D, TCP Tests

Finally, Table 5.1 shows the average network jitter observed for different data transfer sizes during the UDP tests. The overall average is 0.0605 milliseconds, which is very low and falls within the expectations for this topology.

Data Size:	1 MB	10 MB	50 MB	500 MB
Jitter (ms):	0.061	0.054	0.065	0.062

Table 5.1 - Results of Switched Network, No MT6D, UDP Tests

5.1.2 Python MT6D

The same set of tests was run again on the switched topology, but this time the Python version of MT6D was running on the two gateways. This setup forces packets travelling from Host A to Host B to be tunneled through MT6D between the two gateways. No other changes were made to the setup. Due to the addition of MT6D, the results of these tests are expected to show inferior performance compared to the tests run with no MT6D.

Figure 5.3 shows the results of the ICMP tests and is similar in layout to the ICMP test figure in the previous section. The Python version performed very poorly in the ping flood tests,

with over 90% packet loss and maximum latencies as high as 30 seconds. Observation of the test in progress showed that the packet loss occurred consistently throughout the test, indicating that the Python version simply cannot keep up with the number of packets traveling through it, even though the ping packets are only 64 bytes in size. This result indicates that the Python version is not suited to handling large volumes of packets.

Closer inspection of the standard ping test shows a more reasonable, though still problematic, packet loss of 2.32% and an average latency of 103 milliseconds. Observation of the tests showed that most of the packet loss in this test occurs when Python MT6D rotates its IPv6 addresses, as the method it uses causes packets to be dropped momentarily just after binding the new address. These results also suggest that Python MT6D is unsuitable for use in real-time data applications, such as VoIP. The average latency of the standard ping test (103 ms) is acceptable according to the ITU's recommendation, but the much higher latency in the flood tests indicate that the average latency would likely rise when under stress from constant real-time data packets.

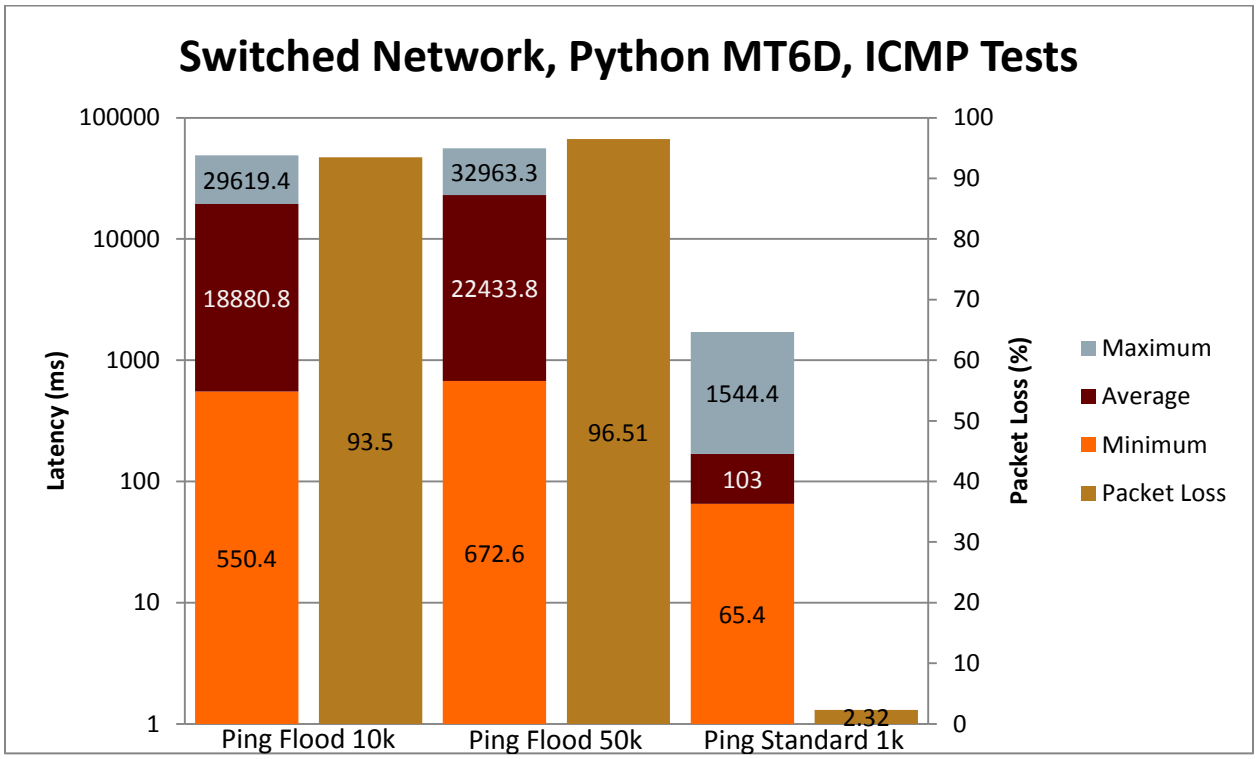

Figure 5.3 - Results of Switched Network, Python MT6D, ICMP Tests

Figure 5.4 shows the results of the TCP tests. At first glance, these results appear similar to, if not better than, the results with no MT6D. However, this figure's vertical axis units are kilobytes per second, rather than megabytes per second as in the previous section. The overall average throughput for Python MT6D is 24.072 kilobytes per second, three orders of magnitude less than with no MT6D. This calculation is a very significant decrease and would be readily noticed by any user transferring data through MT6D. Also, the throughput decreases with larger file sizes; the average for the 500 megabyte test is 18.34 kilobytes per second.

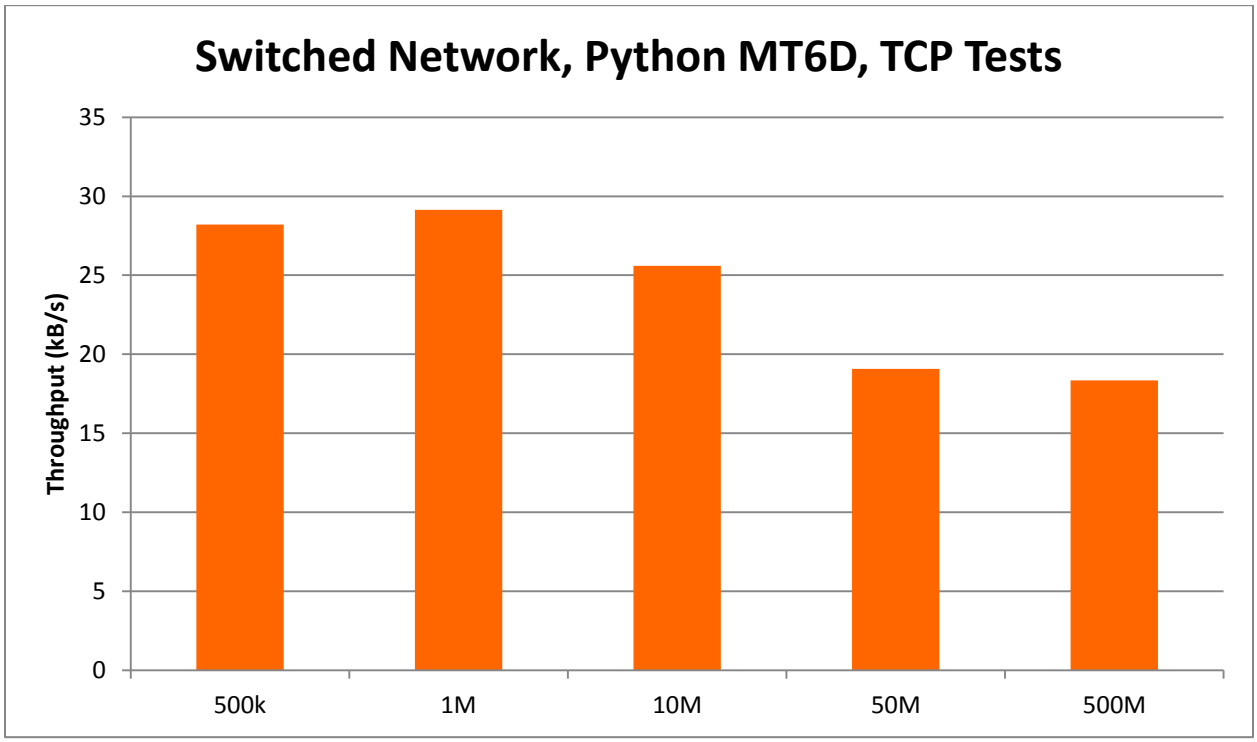

Figure 5.4 - Results of Switched Network, Python MT6D, TCP Tests

Finally, Table 5.2 shows the average network jitter observed for different data transfer sizes during the UDP tests. The overall average is 15.64 milliseconds, which is much greater than the baseline established with no MT6D, indicating that Python MT6D has introduced fluctuations in the network delay.

Data Size:	1 MB	10 MB	50 MB	500 MB
Jitter (ms):	15.573	15.647	15.672	15.667

Table 5.2 - Results of Switched Network, Python MT6D, UDP Tests

5.1.3 C MT6D

Once again, the same set of tests was run on the switched network topology, but C MT6D was used in this testing on the two gateway devices. No other changes were made to the testing setup. These results are expected to show much better performance than the Python version and approximate the baseline established with no MT6D.

Figure 5.5 shows the results of the ICMP tests. These results are significantly improved over the Python version and track the baseline results very closely. The minimum and average values have increased by approximately 0.2 milliseconds over the baseline, while the maximum values have increased 2-5 milliseconds over the baseline. Two of the tests, the 50,000 packet ping flood and the 1000 packet standard ping, show minimal packet losses. These losses occurred in only one iteration of each test and could not be correlated with any other events in the test. In contrast to Python MT6D, these results indicate that C MT6D could easily carry real-time data.

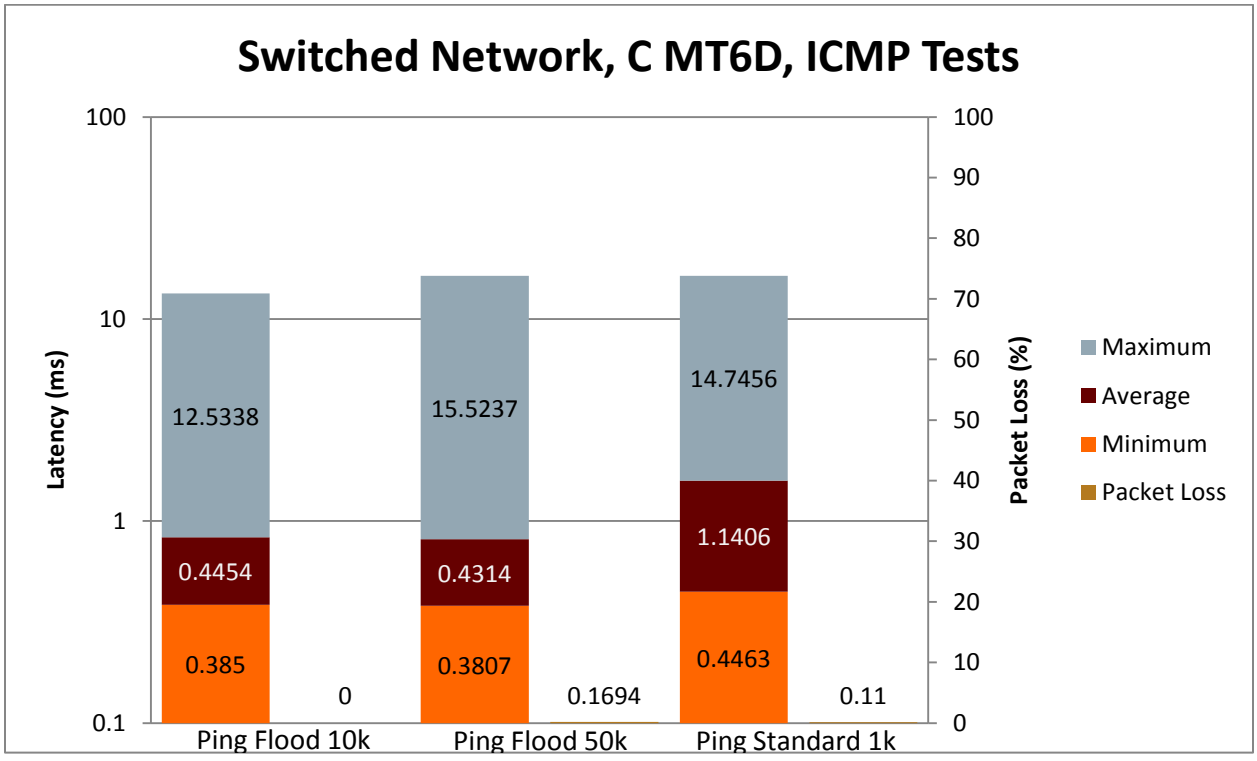

Figure 5.5 - Results of Switched Network, C MT6D, ICMP Tests

Figure 5.6 shows the results of the TCP tests. Similar to the results for the baseline tests, the units of the vertical axis are again shown in megabytes per second. The performance of C

MT6D shown here is two to three orders of magnitude better than Python MT6D, and only slightly less than the baseline. The overall average throughput for C MT6D is 10.57 megabytes per second, compared with 24.10 kilobytes per second for Python MT6D and 19.58 megabytes per second for the baseline. The closest result between the baseline and C MT6D is in the 500 megabyte test, where the baseline average is 15.41 megabytes per second and the C MT6D average is 11.55 megabytes per second. In addition, the throughput in the C MT6D results increases with larger data sizes – due to the influence of the startup delays (neighbor discovery and initial session key exchange) being reduced by the length of the data transfer. This result is especially obvious in the 500 kilobyte data transfer, which only takes 50 milliseconds to complete.

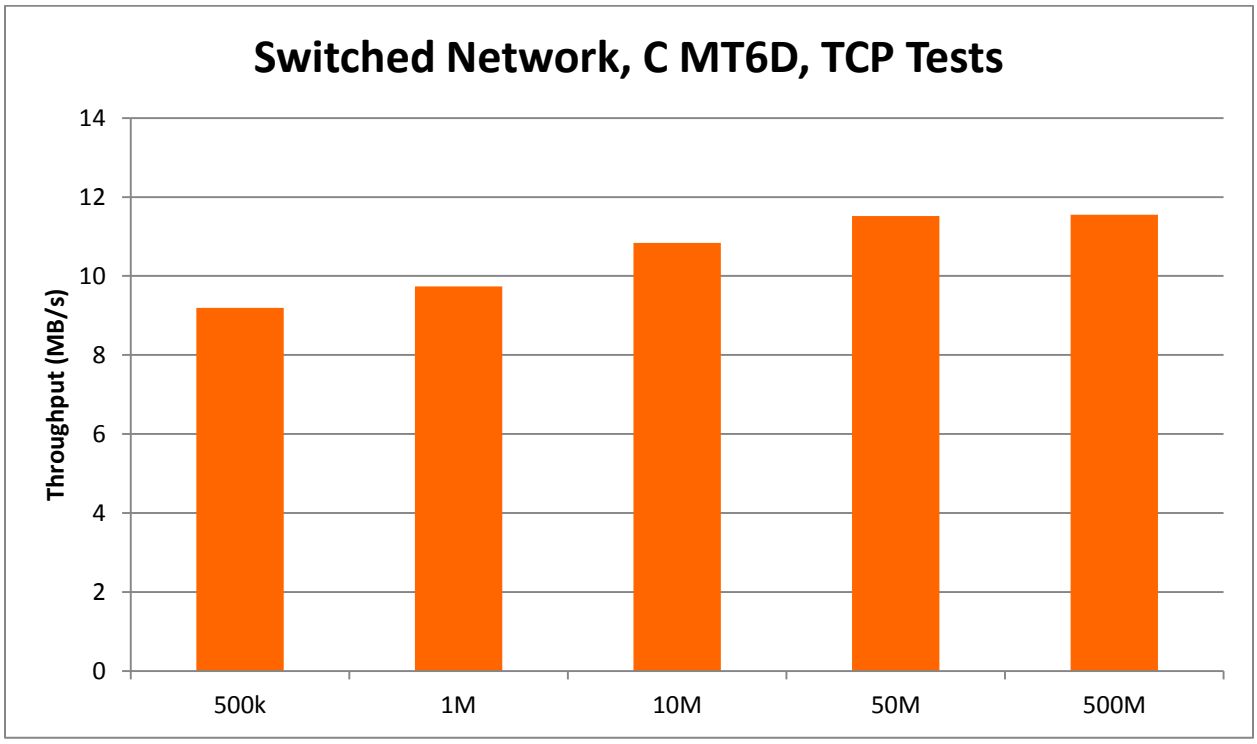

Figure 5.6 - Results of Switched Network, C MT6D, TCP Tests

Finally, Table 5.3 shows the average network jitter observed for different data transfer sizes during the UDP tests. The overall average is 0.08875 milliseconds, which, while 28 microseconds greater than the baseline, is still very low, indicating minimal fluctuations.

Data Size:	1 MB	10 MB	50 MB	500 MB
Jitter (ms):	0.081	0.085	0.094	0.095

Table 5.3 - Results of Switched Network, C MT6D, UDP Tests

5.1.4 Host Performance

The following results show the effect that running MT6D had on the host serving as the gateway. As detailed in the previous chapter, the average CPU usage, memory usage, and kernel/user time were collected for comparison. MT6D is likely to be deployed on a resource constrained system, such as a router, which would require that it be very efficient in its resource use. In the next sections, the host metrics mentioned are compared between the Python and C versions.

Figure 5.7 shows a comparison of CPU usage and memory usage between the Python and C versions. Data from the ping flood and standard ping tests are shown. These two tests were chosen because the ping flood generates an extremely large volume of constant traffic, while the standard ping only generates one small packet each second. Therefore, results from the ping flood show how resources are used by the two versions under heavy load, and results from the standard ping show how resources are used by the two versions while they are mostly idle.

As shown, a big difference exists between the two versions. Under heavy load, the Python version uses almost 32% of the CPU, while the C version uses only 12%. As expected, the CPU usage is much less under the standard ping test, with the Python version using 2% of the CPU and the C version barely registering at 0.01%. Memory usage is different, with usage in both versions mostly constant between the two tests. The Python version uses around 4 megabytes while the C version uses around 60 kilobytes. This result indicates that, once both versions have started, their memory usage remains mostly constant, regardless of load. These results show that the C version is better suited for a small, resource constrained system.

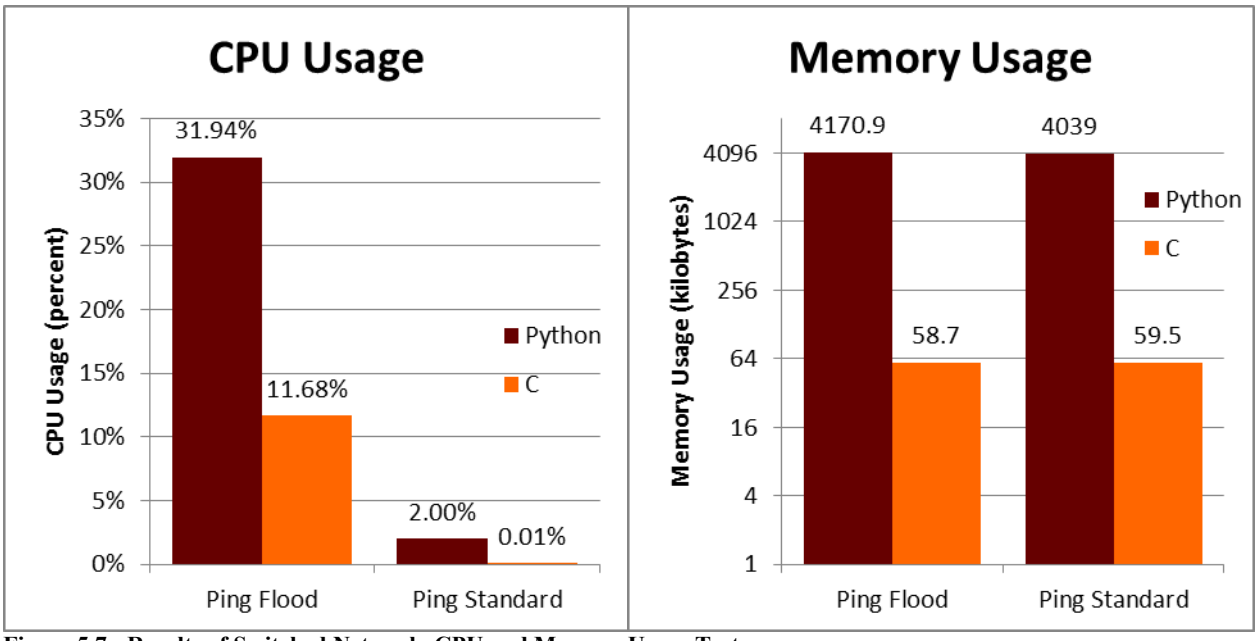

Figure 5.7 - Results of Switched Network, CPU and Memory Usage Tests

The results of the third host metric tests, kernel/user time, are shown in the next figures with the ping flood test first (Figure 5.8). In the ping flood test, most of the work involves copying and manipulating fields in packet data. The figure on the right shows that the Python version spends most of its execution time (81.7%) executing code in the Python library, while only 4.97% of execution time is spent in the Linux kernel. By contrast, the figure on the left shows that 79.4% of the C version's execution time is spent in the kernel and only 3% is spent in the MT6D executable.

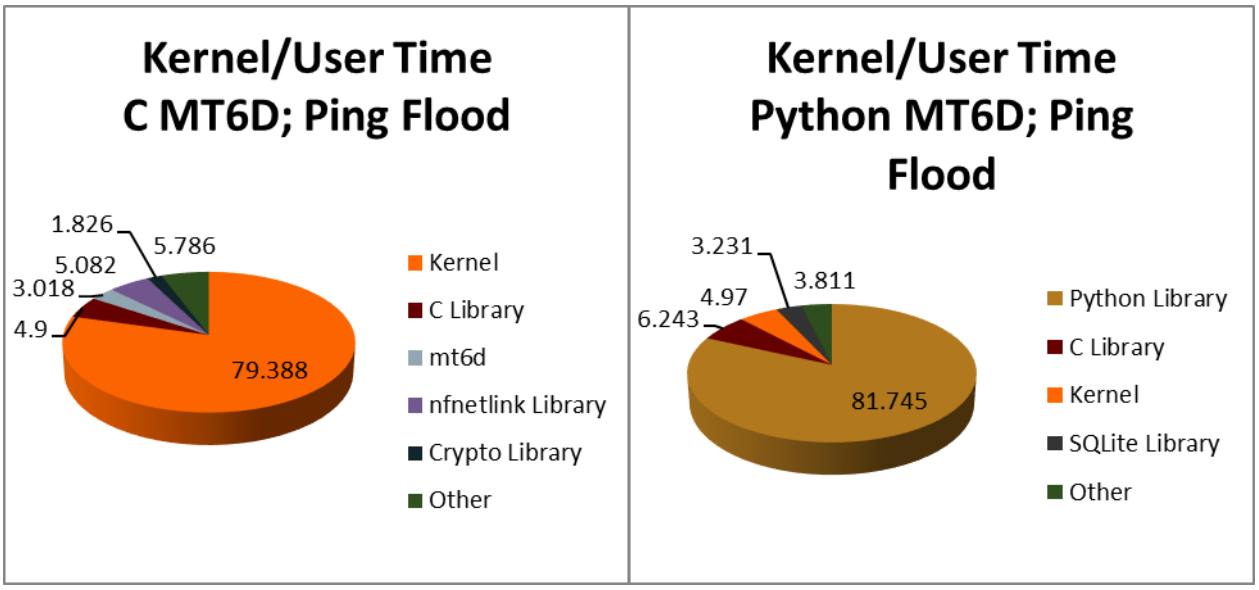

Figure 5.8 - Results of Switched Network, Kernel/User Time, Ping Flood Tests

Figure 5.9 shows the same information as in Figure 5.8 for the standard ping test, when the MT6D gateways are mostly idle. When idle, most of the work revolves around binding/unbinding IPv6 addresses and using the cryptographic hash function to generate new addresses.

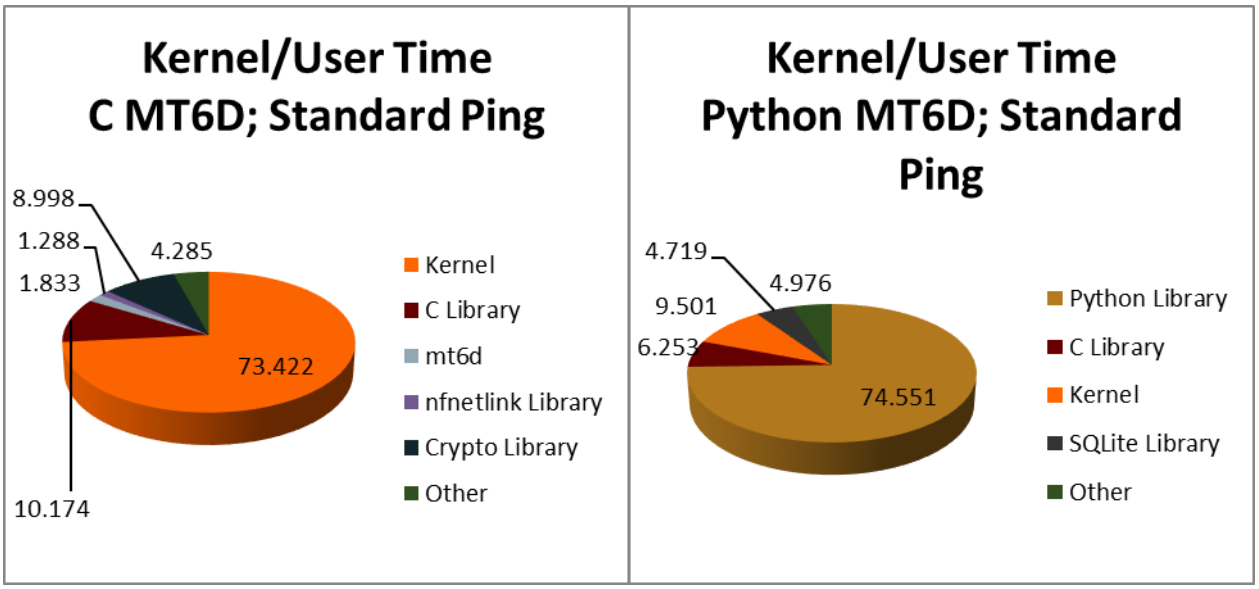

Figure 5.9 - Results of Switched Network, Kernel/User Time, Standard Ping Tests

As these two figures show, the goal of moving most of the processing to kernel code in the C version was met. This goal is at least partially responsible for the improved performance

exhibited by the C version in the other host metrics. These figures also show what other libraries are used most frequently by the two programs. The C version uses functions from the nfnetlink library (used for instructing the kernel to bind/unbind addresses and to add/remove ip6tables rules) and the Crypto library (cryptographic hashing functions). The Python version uses functions from the SQLite library (SQLite is the storage engine used to maintain information about routes and addresses). Both versions make use of utility functions in the C library.

5.2 Routed Topology

The following results were obtained under the routing topology. As discussed in the previous chapter, the routing topology uses a single router between the two MT6D gateways. The purpose of routed topology is to expose the MT6D traffic to other, unrelated traffic that exists in the switches and router. This traffic can introduce unexpected latencies and packet loss, which MT6D must be able to handle, while not causing a significant negative impact on the overall performance.

5.2.1 ICMP Tests

The following two figures show the results of the ICMP tests. Figure 5.10 shows the average latency for each of the three ICMP tests for no MT6D, Python MT6D, and C MT6D. Figure 5.11 shows the packet loss for the same tests. These results are very similar to the results for the ICMP tests on the switched topology. Again, the Python version's latency and packet loss are significantly higher, while the C version's latency and packet loss approximate the baseline performance. This result indicates that MT6D is not affected by the presence of a router, and the relative performance of Python MT6D versus C MT6D is also not affected by a router.

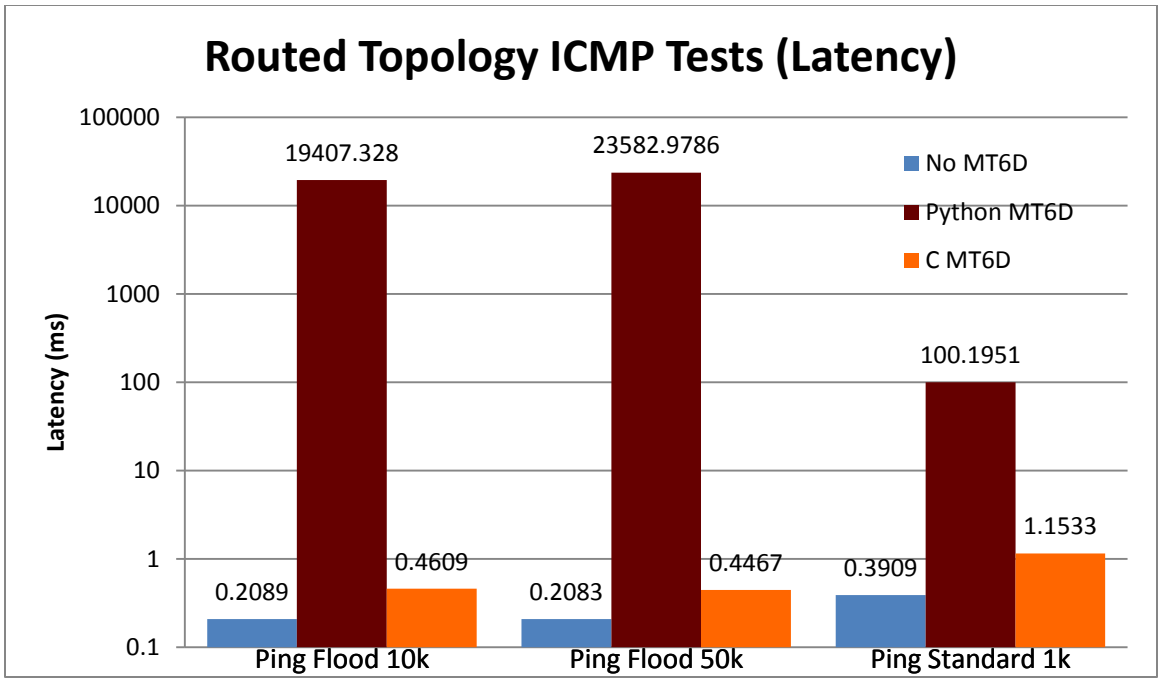

Figure 5.10 - Results of Routed Network, ICMP Latency Tests

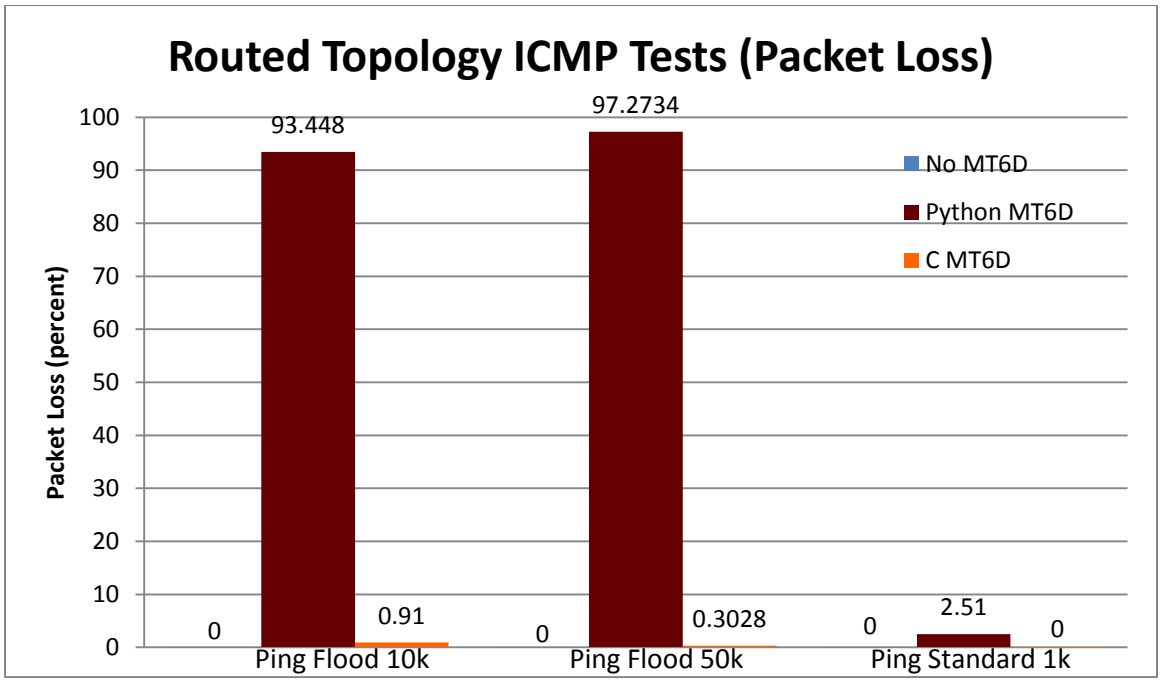

Figure 5.11 - Results of Routed Network, ICMP Packet Loss Tests

5.2.2 TCP Tests

Results of the routed topology TCP tests are shown in Figure 5.12. This figure shows data for no MT6D, Python MT6D, and C MT6D for each data transfer size. As with the

switched topology, the throughput with the Python version is three orders of magnitude less than the baseline and two to three orders of magnitude less than the C version. Once again, the absolute throughput for each case is almost identical to the results in the switched topology, showing that it has minimal effect on MT6D.

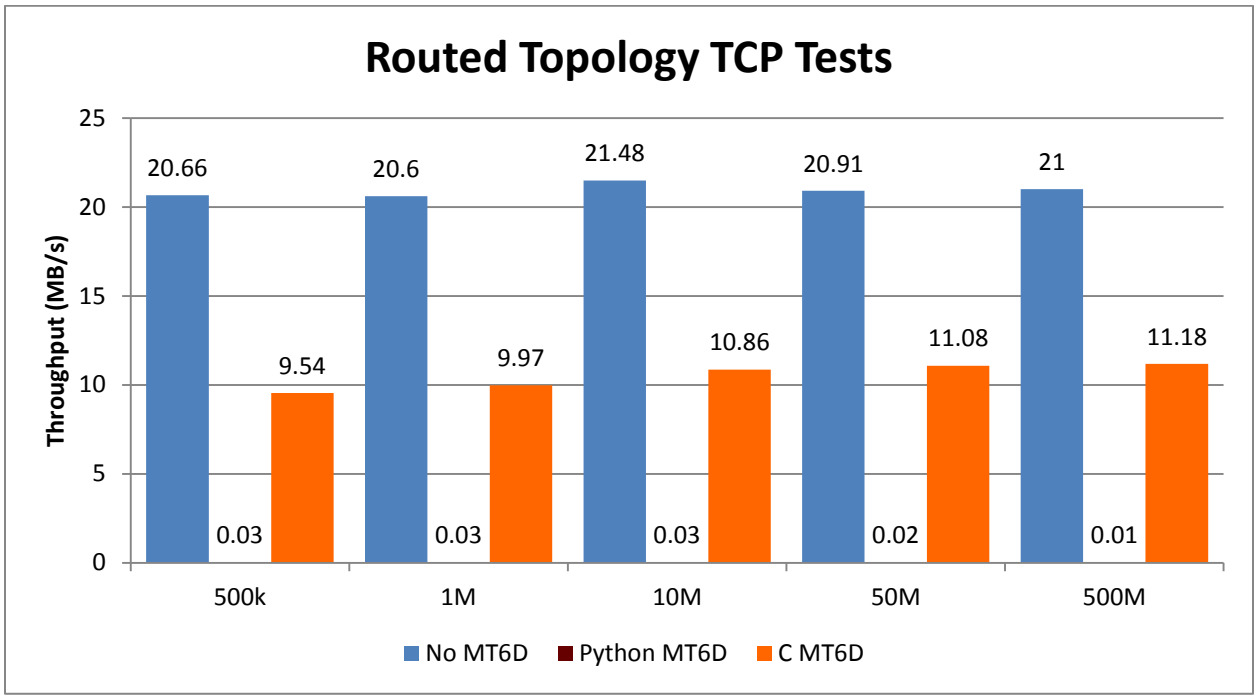

Figure 5.12 - Results of Routed Network, TCP Tests

5.2.3 Host Performance

Figure 5.13 compares CPU usage and memory usage for Python MT6D and C MT6D on the routed topology. As expected, the figure does not show much change from data for the switched topology. Adding a router between the gateways has not affected the load experienced by MT6D, so the resources required from the host have also not changed.

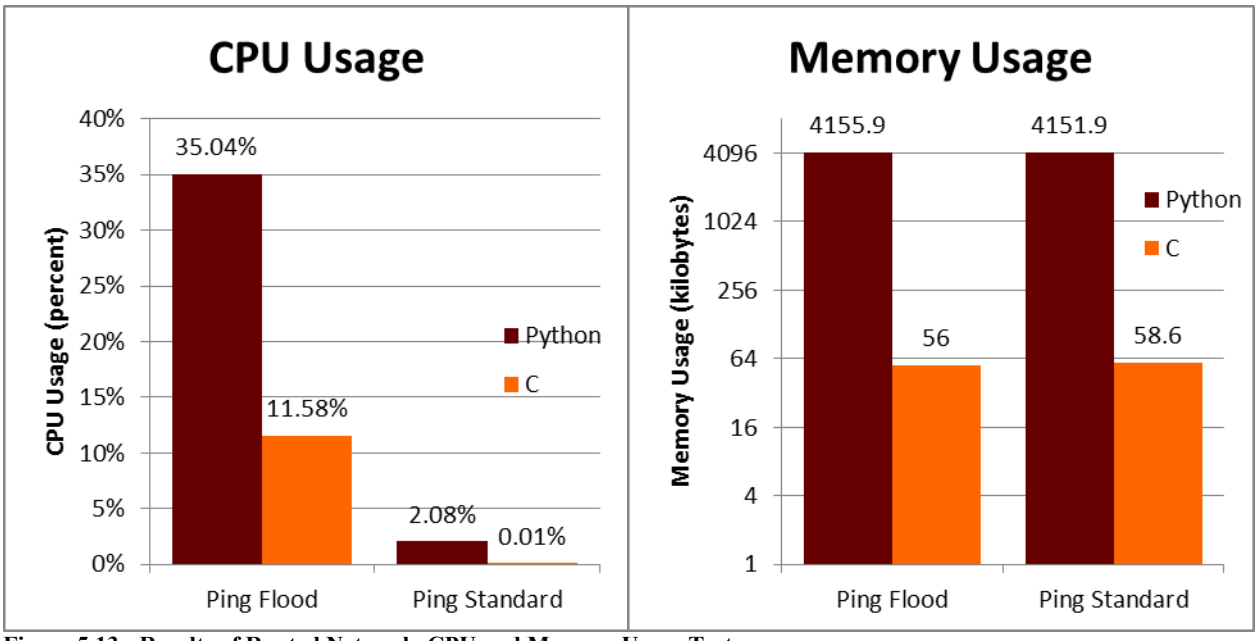

Figure 5.13 - Results of Routed Network, CPU and Memory Usage Tests

Figure 5.14 and Figure 5.15 show the kernel/user time comparison for Python MT6D and C MT6D during the ping flood and standard ping tests. As with other host metrics, these results are very similar to those obtained on the switched topology and confirm that the addition of the router does not affect the host resources required by MT6D.

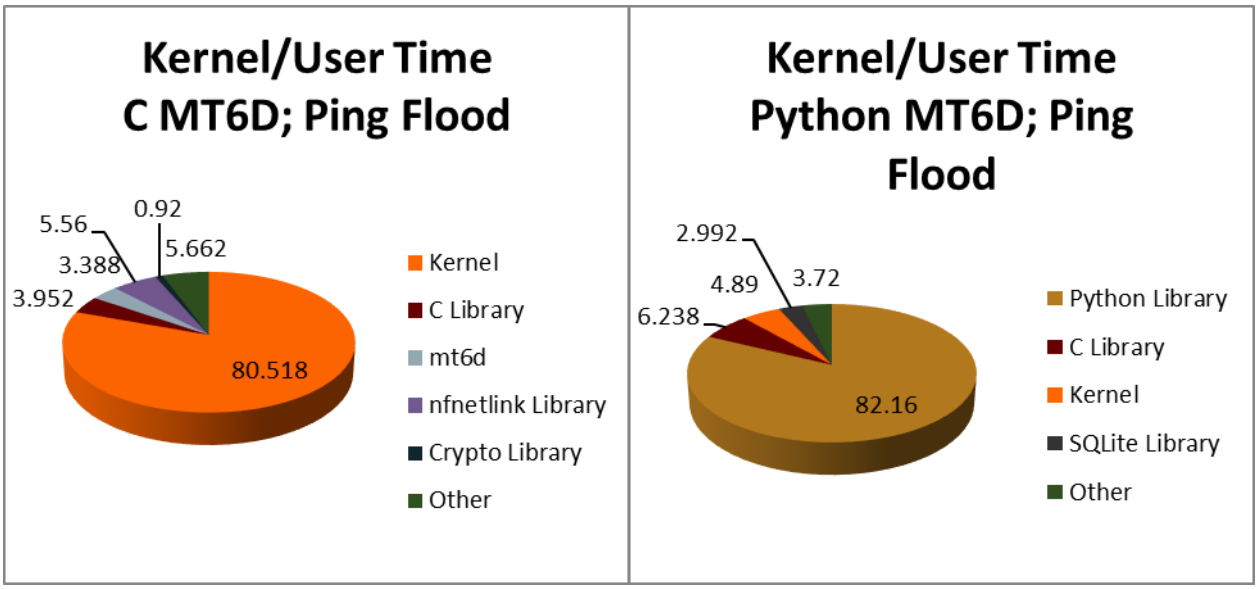

Figure 5.14 - Results of Routed Network, Kernel/User Time, Ping Flood Tests

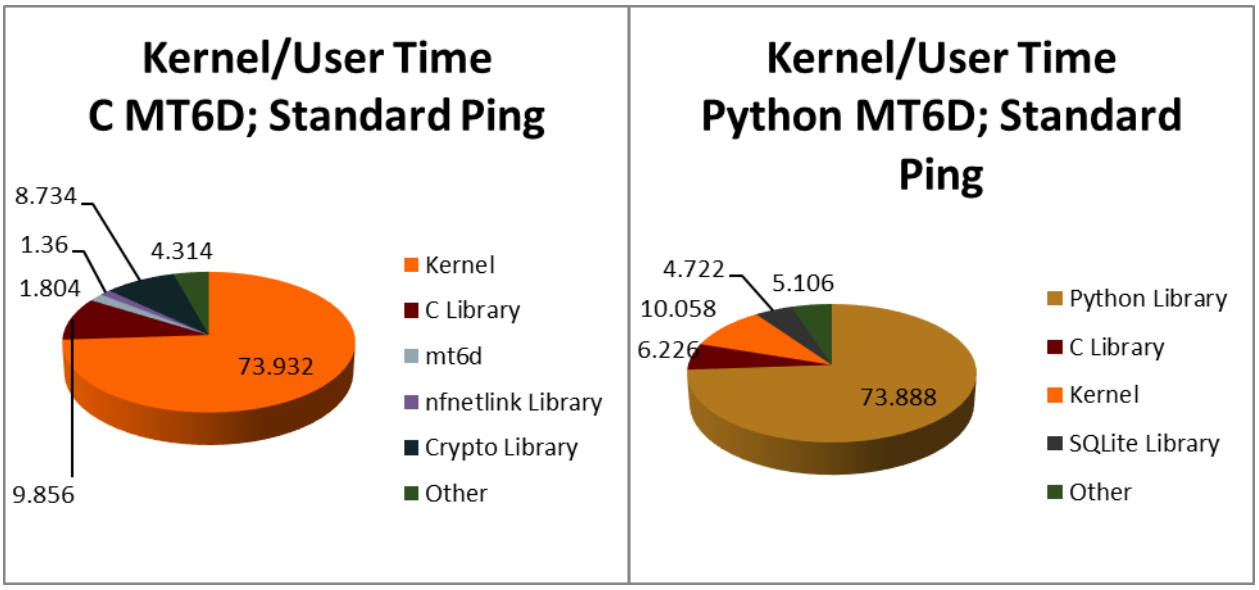

Figure 5.15 - Results of Routed Network, Kernel/User Time, Standard Ping Tests

5.3 Tunneled Topology

As described in the previous chapter, tunneled topology is the most complicated and routes traffic through a 6in4 gateway in Seattle, Washington. This configuration causes traffic to be tunneled inside IPv4 traffic for part of its trip, and it must cross the United States twice to make the journey. This test is designed to validate the ability of MT6D to be tunneled inside another protocol and to coexist with large amounts of unrelated traffic on the Internet. However, due to the nature of this topology, performance is expected to be much lower for this topology, even without MT6D.

5.3.1 ICMP Tests

Figure 5.16 shows the latency of tunneled topology for tests using no MT6D, Python MT6D, and C MT6D. While this chart bears some similarities to the previous latency charts, some important differences exist. First, the latency of the baseline test (with no MT6D) has increased, due to the nature of tunneled topology. Like previous tests, the Python version is unable to cope with ping floods and shows very high latencies for those tests. The C version has only slightly higher latencies than the baseline for all the tests. The other major difference is the latency of the Python version for the standard ping test. The Python version's latency is only about 20 milliseconds higher than the baseline. This result is due to the network topology

introducing a delay, which allows the Python version to "catch up" with the baseline and the C version.

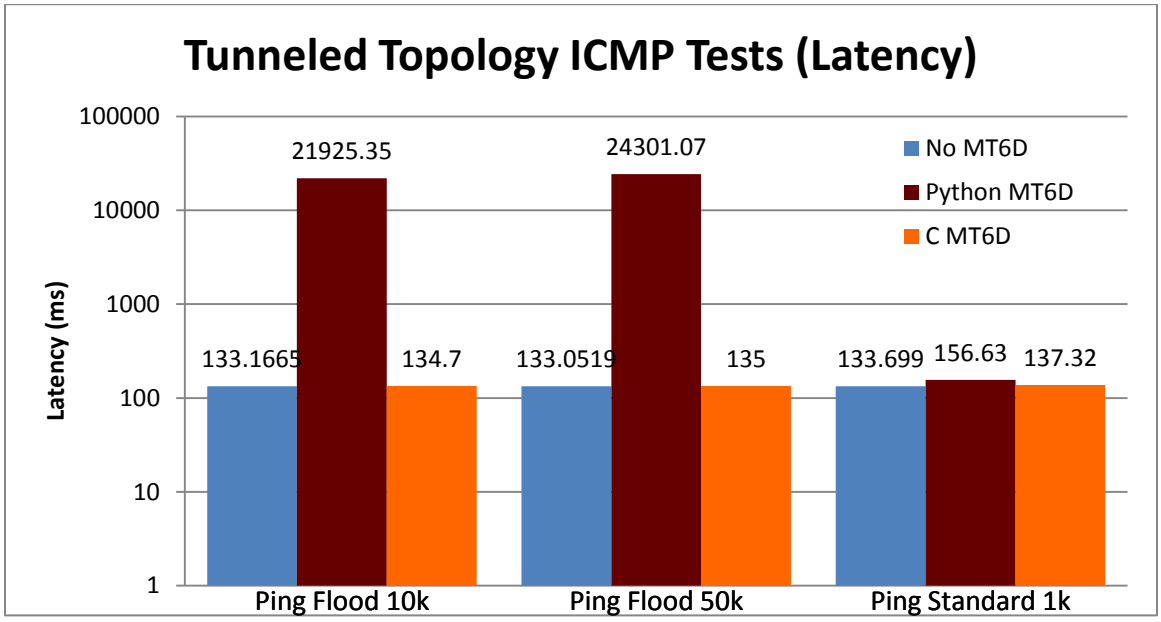

Figure 5.16 - Results of Tunneled Network, ICMP Latency Tests

Figure 5.17 shows the packet loss recorded during the ICMP tests. This figure is almost identical to results for the other topologies, except that the baseline now shows a small amount of packet loss.

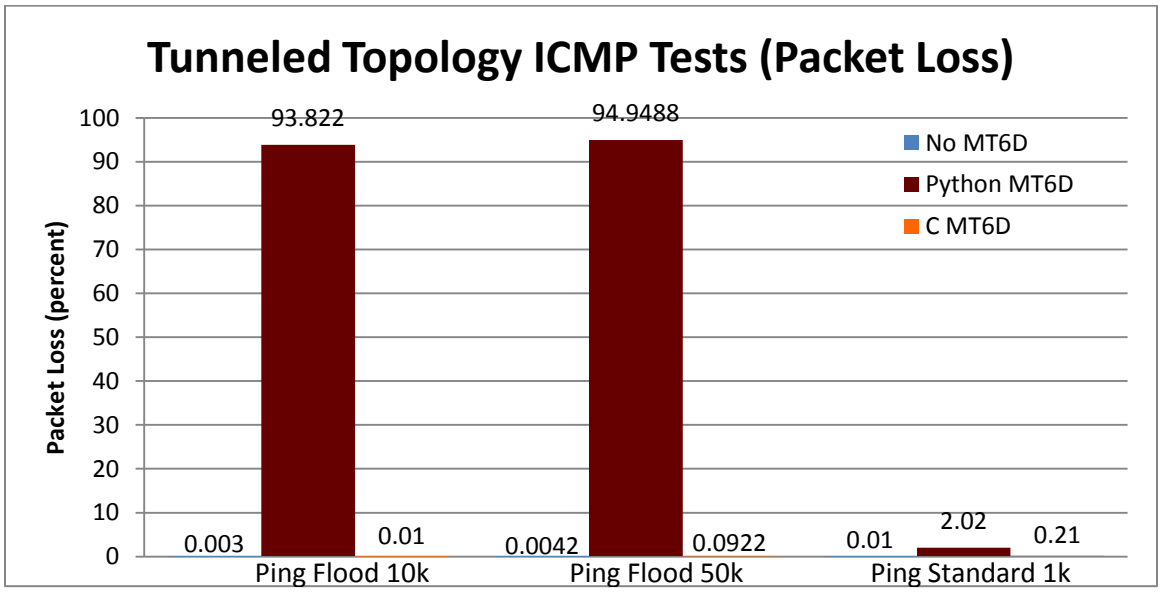

Figure 5.17 - Results of Tunneled Network, ICMP Packet Loss Tests

5.3.2 TCP Tests

Figure 5.18 shows the results of the TCP tests for tunneled topology. The Python version results are the same as for previous topologies. The baseline results report a much lower throughput for this topology than for the other topologies. The reduced throughput is less than the maximum that the C version attained in previous results; consequently, in these results, the throughput of the C version is almost equivalent to the baseline. These results also show that the tunneled network topology is better suited to larger file transfers than small ones.

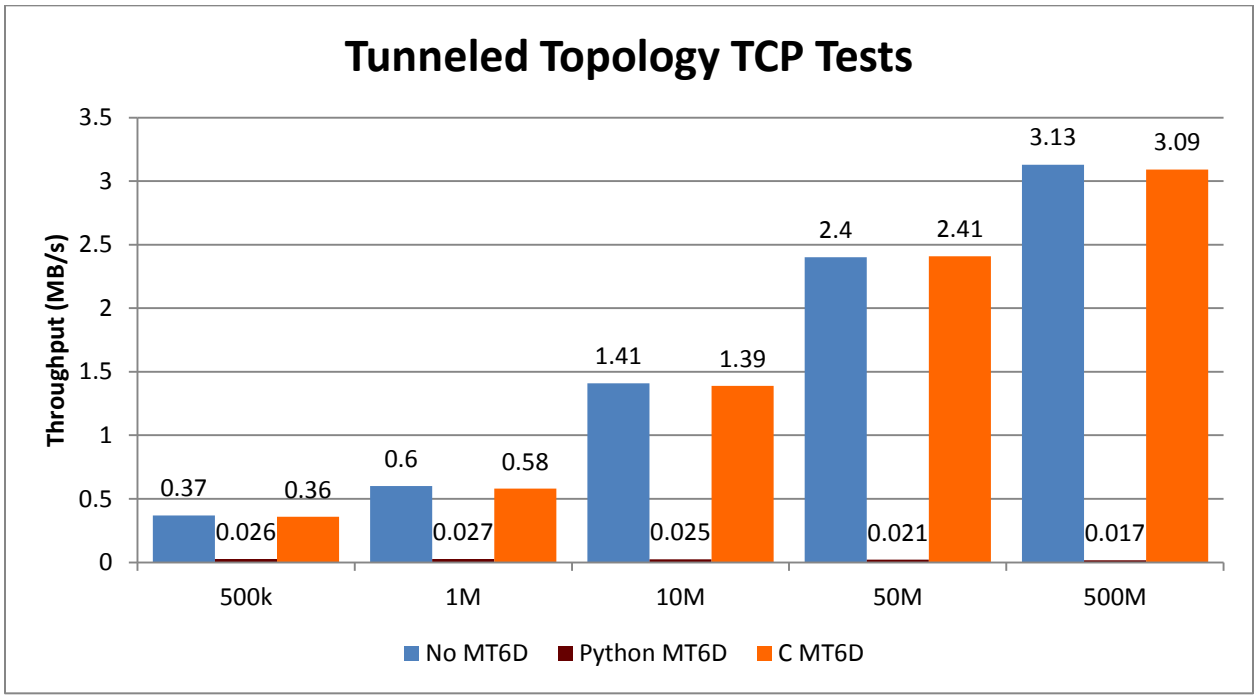

Figure 5.18 - Results of Tunneled Network, TCP Tests

5.3.3 Host Performance

Figure 5.19 shows CPU usage and memory usage for the Python version and C version on tunneled topology. Memory usage of both versions and CPU usage of the Python version are almost identical to previous results. However, CPU usage of the C version under load (the ping flood test) has dropped significantly, from about 11% in previous tests to 1.73%. This reduction is due to the tunneled network topology delivering packets at a much slower rate, effectively throttling the MT6D connection. The lower rate is still faster than the Python version can

deliver, so CPU usage of the Python version remains the same. However, the rate is less than the C version can deliver, so the C version is idle some of the time during this test.

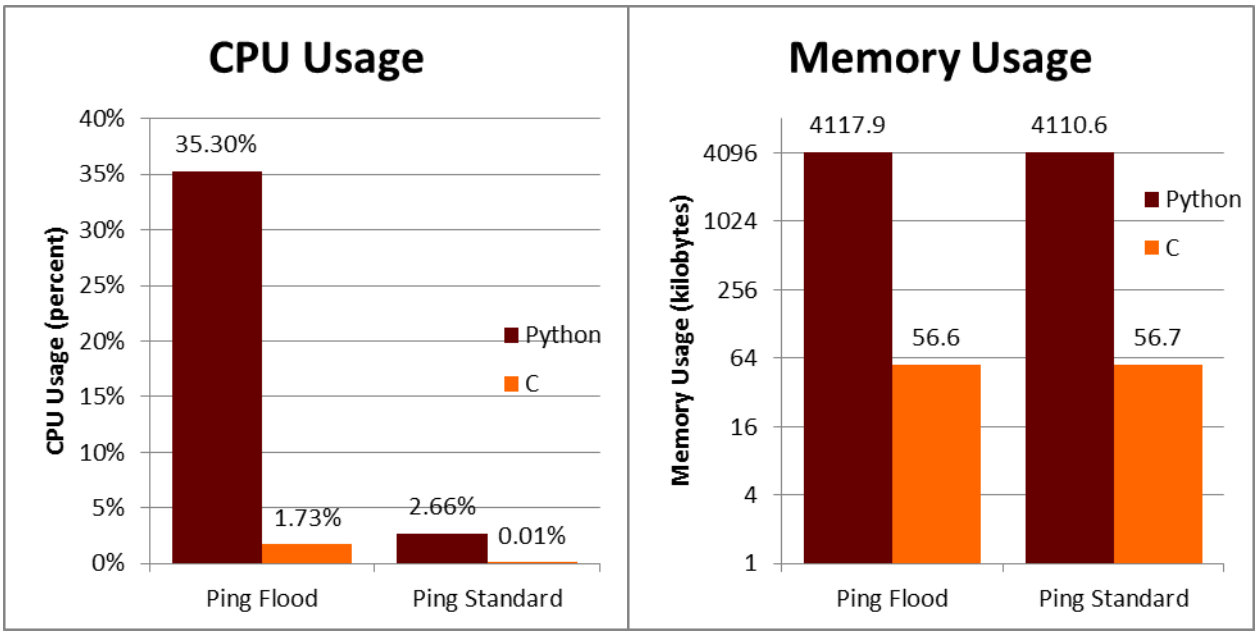

Figure 5.19 - Results of Tunneled Network, CPU and Memory Usage Tests

Figure 5.20 and Figure 5.21 show the kernel/user time for tunneled topology. As expected, these figures are very similar to the same metrics for the other topologies, with one small difference. If results for the C version in the other topologies are closely examined, the conclusion is that time spent in the kernel is a lower percentage for the standard ping test, since less time is spent handling packets in that test. In these results, the same reduction in kernel time is observed in the ping flood results, due to the lower volume of packets.

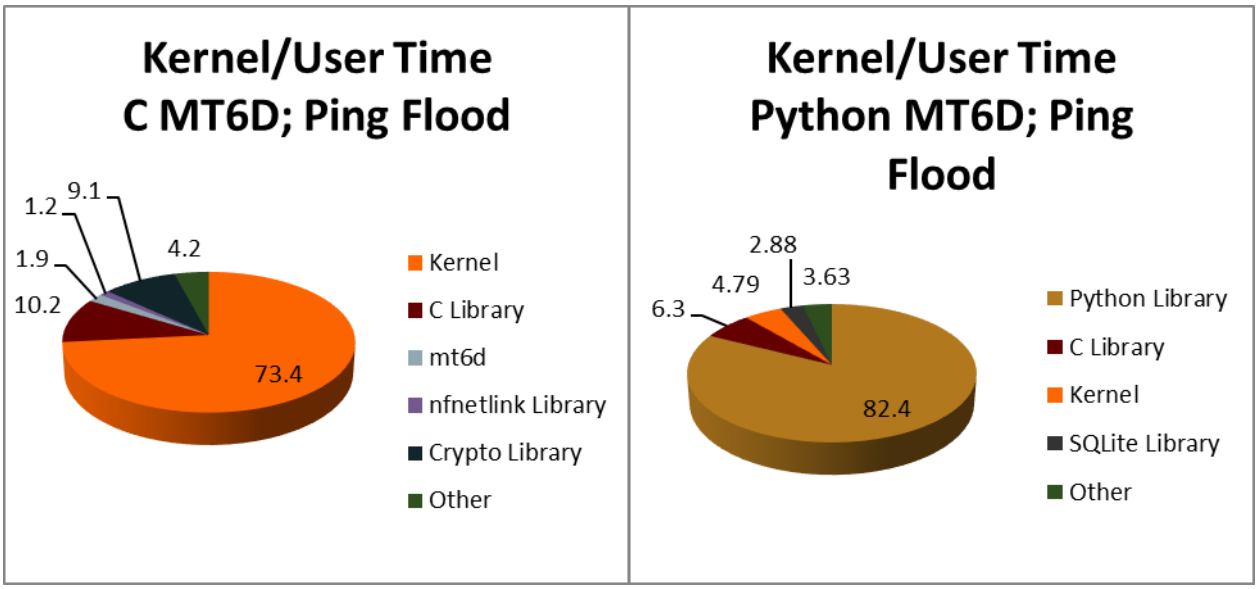

Figure 5.20 - Results of Tunneled Network, Kernel/User Time, Ping Flood Tests

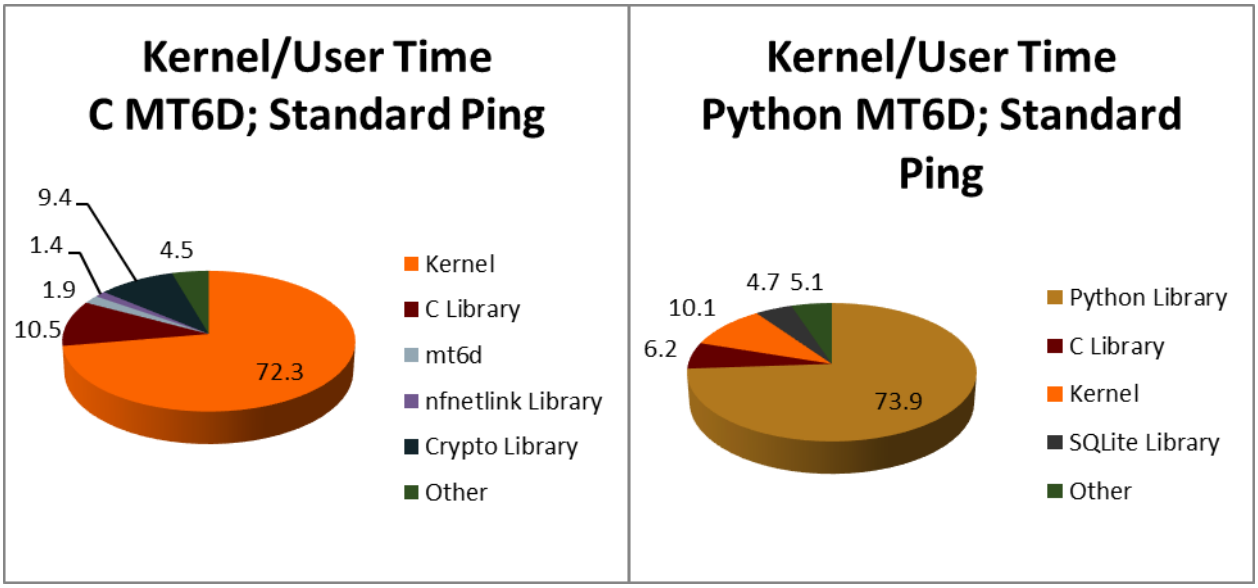

Figure 5.21 - Results of Tunneled Network, Kernel/User Time, Standard Ping Tests

5.4 Summary

This chapter presents the results of the live network tests of MT6D. Tests performed were designed to test the ability of MT6D to carry different types of traffic and to test the design of the C version to determine if the goal of improved performance was met. Three network topologies, switched, routed, and tunneled, were used during the test to expose MT6D traffic to varied network conditions, paths, and unrelated traffic. Each test and topology was tested with no MT6D to establish a baseline, and then with Python MT6D and C MT6D to compare the

performance of the two versions with the baseline and with each other. For each test, network metrics – such as latencies, packet loss, throughput, and jitter – and host metrics – including CPU usage, memory usage, and kernel/user time – were collected, compared, and analyzed.

Figure 5.22, Figure 5.23, and Figure 5.24 show the combined results for switched topology for no MT6D, Python MT6D, and C MT6D, showing latencies, packet loss, and throughput. These results show that C version consistently delivers lower latencies and packet losses and higher throughput than the Python version, while performing very closely to the baseline. These results validate the design choices made in Chapter 3.

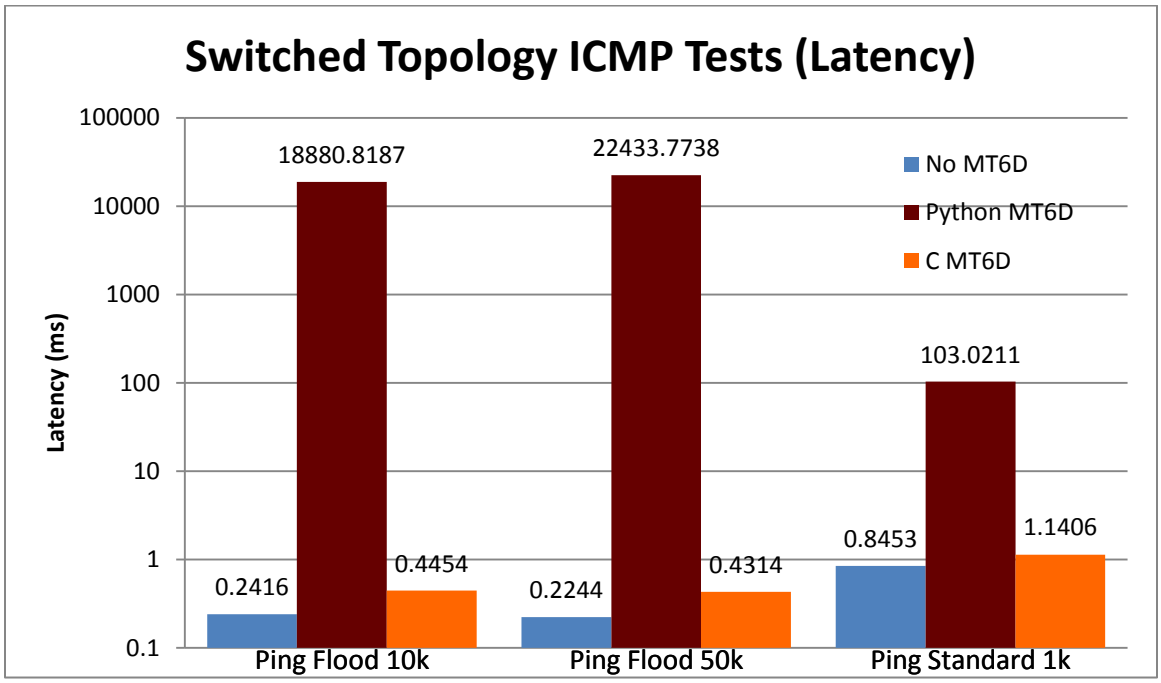

Figure 5.22 - Results of Switched Network, ICMP Latency Tests

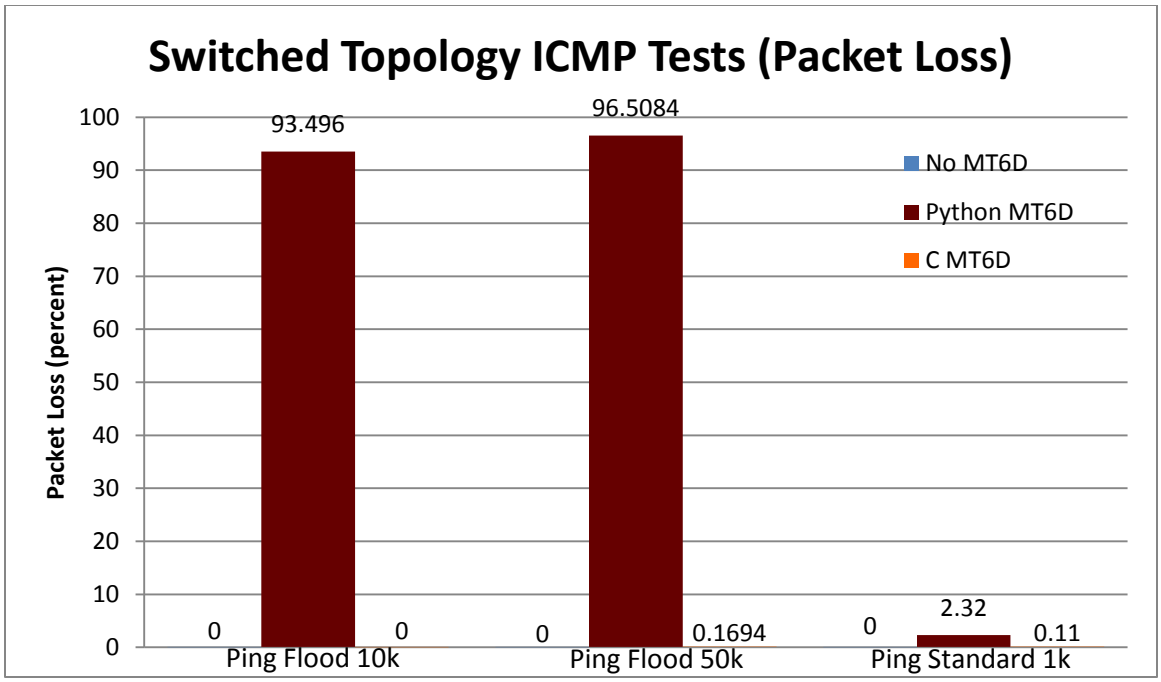

Figure 5.23 - Results of Switched Network, ICMP Packet Loss Tests

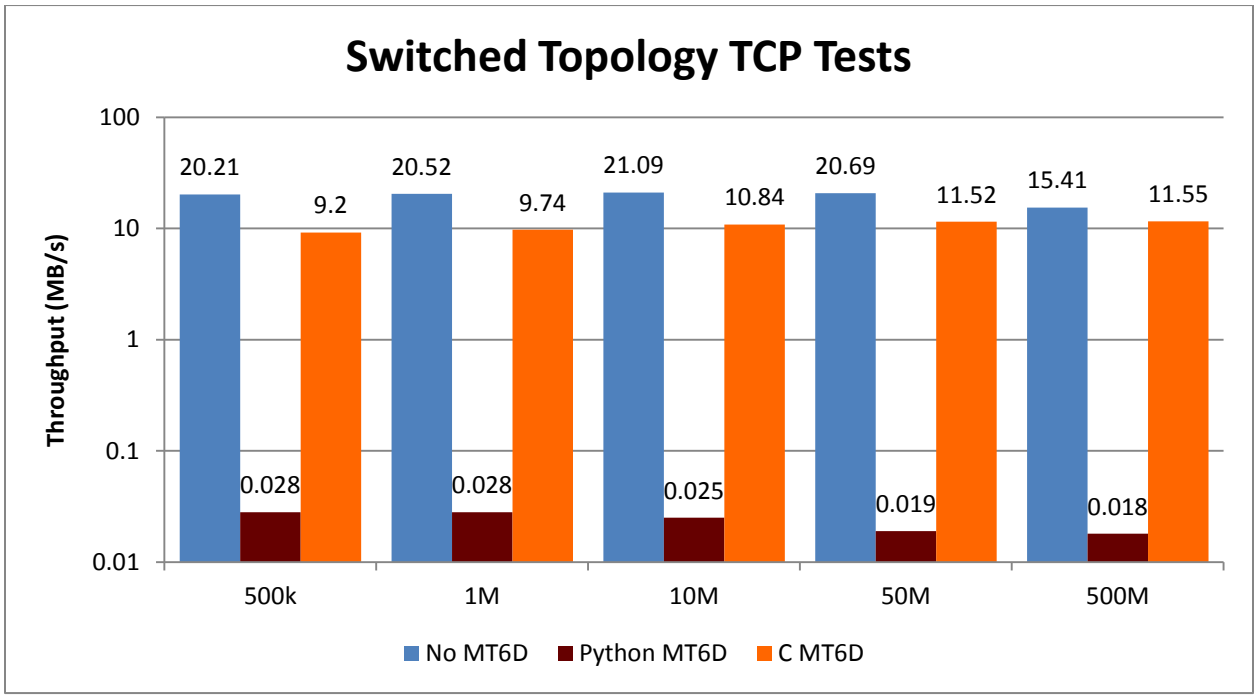

Figure 5.24 - Results of Switched Network, TCP Tests

Chapter 6: Conclusion

The primary goal of this work was to identify specific parts of the Python version of MT6D that could be used as improvements in the development of the C version. The largest area identified for improvement was the way in which packets are captured in the Python version, using libpcap. In the C version, packet capturing is done by having the Linux kernel identify packets using ip6tables and copying them to the MT6D program using netfilter queues. This strategy is part of a broader effort to move as much work as possible to the kernel from the MT6D program. Other examples of moving work to the kernel include binding remote addresses locally so that the network stack will generate neighbor discovery packets and using libraries to send address and table commands to the kernel directly, rather than calling external programs such as 'ip' and 'ip6tables .'

Another goal in the C version was better utilization of multiprocessing capabilities. The C version maintains one thread to handle ICMP error messages, one thread to handle address mutations, and one thread for each pair of hosts communicating via MT6D. The multithreaded design also avoids locks, instead using atomic operations whenever shared data must be modified.

This document details the steps taken to meet these goals. Chapter 2 describes the MT6D protocol and its implementation in Python. Flow of the Python program is described and specific concerns noted in its efficiency and operations. Chapter 3 describes the steps that would be taken in the C version to mitigate the concerns in the Python version. The chapter also describes in detail how the C version works and documents an example packet exchange. Chapter 4 lists various topologies, scenarios, and tests that were performed to test the C version and the Python version to evaluate the effectiveness of the design choices. Finally, Chapter 5 presents the results of the tests and shows how the design choices are validated and delivered significant performance improvement.

6.1 Future Work

Since design choices made for the C version have been validated by tests and have resulted in significant performance improvements, they provide some insight into future work

that could be done to improve MT6D even further. As this work demonstrates, closer integration with the kernel provides performance increases for an MTD. Future work should focus on moving MT6D closer to the Linux kernel, either as a kernel module or even integrating it with the network stack. Once a part of the kernel, MT6D would be in a better position to capture, modify, and control the disposition of packets without requiring that the packet first be copied to user space.

Another possible future direction would be to integrate MT6D into hardware. On a small, embedded system, such as a router, an MTD could be made very efficient if it were implemented on a Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC). In this way, the MTD could control packets before they are even delivered to the operating system. An example of this type of integration can be seen with cryptographic functions – historically crypto operations were implemented only in software but are now available as hardware crypto chips [42] and even as instructions in some processors [43].

6.2 Concluding Thoughts

Results of the tests that were conducted in the third phase of this work show significant performance gains made by the C version of MT6D. While the bandwidth available using the C version is less than the bandwidth available without MT6D, it is much more than is available with the Python version. Also, network latency and jitter decrease and are very close to the values without MT6D. The C version also drops very few packets compared to the Python version. Execution time of the program is spent more efficiently by using kernel code to do much of the work in the C version, rather than running code from the Python library. The C version uses less memory and less processor time to accomplish its work. These results show the performance gains made possible by converting MT6D to the C language.

Moving target defenses require significant and frequent modifications to system parameters to create successful entropy and act as a viable defense. System architecture independent languages, such as Python and Java, have many benefits that enhance rapid development of a working prototype and allow for code portability and reuse. However, the operations of a MTD that are required to create entropy are too resource intensive for these languages. By using compiled languages and operating system specific features, MTDs such as

MT6D can be deployed successfully in many different types of network systems, including resource constrained environments.

References

[1] Brian Krebs. Sources: Target Investigating Data Breach. Available at: http://krebsonsecurity.com/2013/12/sources-target-investigating-data-breach, accessed on 8 July 2014.

[2] Ewen MacAskill and Gabriel Dance. NSA Files: Decoded. Available at: http://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded, accessed on 8 July 2014.

[3] Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. The Blind Man's Bluff Approach to Security Using IPv6. *IEEE Security & Privacy*, vol. 10, no. 4, pp. 35-43, July-Aug. 2012.

[4] J. Postel. Internet Protocol. RFC 791 (Internet Standard), September 1981. Updated by RFCs 1349, 2474, 6864.

[5] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6). RFC 1883 (Proposed Standard), December 1995. Obsoleted by RFC 2460.

[6] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6). RFC 2460 (Draft Standard), December 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112.

[7] Central Intelligence Agency. The World Factbook. Available at: https://www.cia.gov/library/publications/the-world-factbook/geos/xx.html, accessed on 3 July 2014.

[8] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401 (Proposed Standard), November 1998. Obsoleted by RFC 4301.

[9] S. Thomson, T. Narten and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), September 2007.

[10] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006. Updated by RFCs 5952, 6052, 7136, 7346, 7371.

[11] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins and M. Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315 (Proposed Standard), July 2003. Updated by RFCs 4361, 5494, 6221, 6422, 6644, 7083, 7227.

[12] Matthew Dunlop, Stephen Groat, Randy Marchany, and Joseph Tront. The Good, the Bad, the IPv6. In *Ninth Annual Conference on Communication Networks and Services Research (CNSR 2011)*, Ottawa, Canada, May 2011.

[13] Stephen Groat, Matthew Dunlop, Randy Marchany, and Joseph Tront. What DHCPv6 Says About You. In *the World Congress on Internet Security (WorldCIS-2011)*, February 2011.

[14] Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. MT6D: A Moving Target IPv6 Defense. In *the 2011 Military Communications Conference (MILCOM)*, pages 1321-1326, Baltimore, Maryland, November 2011.

[15] William Stallings. *Cryptography and Network Security*. Prentice Hall, 2011.

[16] J. Postel. User Datagram Protocol. RFC 768 (Internet Standard), August 1980.

[17] T. Narten, E. Nordmark, W. Simpson and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard), September 2007. Updated by RFCs 5942, 6980, 7048.

[18] A. Conta, S. Deering and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443 (Draft Standard), March 2006. Updated by RFC 4884.
[19] libpcap. Available at: http://www.tcpdump.org/, accessed on 29 September 2011.
[20] Harald Welte and Pablo Neira Ayuso. The netfilter.org "iptables" project. Available at: http://www.netfilter.org/projects/iptables/, accessed on 27 August 2011.
[21] Jozsef Kadlecsik. IP sets. Available at: http://ipset.netfilter.org/, accessed on 27 August 2011.
[22] SQLite. Available at: http://www.sqlite.org/, accessed on 2 July 2014.
[23] David Beazley. Understanding the Python GIL. Presented at PyCon 2010, February 2010. Available at: http://www.dabeaz.com/python/UnderstandingGIL.pdf, accessed on 27 August 2011.
[24] D. Mills, U. Delaware, J. Martin, J. Burbank and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.
[25] International Business Machines Corporation. Compiled versus interpreted languages. Available at: http://www-01.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zappldev/zappldev_85.htm, accessed on 5 July 2014.
[26] Harald Welte and Pablo Neira Ayuso. The netfilter.org "libnetfilter_queue" project. Available at: http://www.netfilter.org/projects/libnetfilter_queue/, accessed on 27 August 2011.
[27] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.
[28] The OpenSSL Project. Libcrypto API. Available at: http://wiki.openssl.org/index.php/Libcrypto_API, accessed on 8 October 2011.
[29] Jan Engelhardt. Xtables-addons. Available at: http://xtables-addons.sourceforge.net/, accessed on 29 April 2012.
[30] Harald Welte and Pablo Neira Ayuso. The netfilter.org "libnfnetlink" project. Available at: http://netfilter.org/projects/libnfnetlink/, accessed on 27 August 2011.
[31] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.
[32] Matthew Dunlop and Stephen Groat, MT6D Python code, unpublished.
[33] Warren Gay. *Linux Socket Programming by Example*. Que Publishing, 2000.
[34] GuruPlug Server. Available at: https://www.globalscaletechnologies.com/t-guruplugdetails.aspx, accessed on 14 July 2014.
[35] The Angstrom Distribution. Available at: http://www.angstrom-distribution.org, accessed on 14 July 2014.
[36] A. Durand, P. Fasano, I. Guardini, and D. Lento. IPv6 Tunnel Broker. RFC 3053 (Informational), January 2001.
[37] International Telecommunication Union. G.114 One-way transmission time, May 2003.
[38] Albert Cahalan. ps source code [version 3.2.8], May 2009. File: ps/output.c.
[39] Gperftools. Available at: https://code.google.com/p/gperftools/, accessed on 27 October 2012.
[40] Linux Kernel Page Monitor. Available at: http://man7.org/linux/man-pages/man5/proc.5.html, accessed on 9 July 2014.

[41] Linux Kernel Performance Counters. Available at: https://perf.wiki.kernel.org/index.php/, accessed on 4 August 2012.
[42] Barco Silex FPGA Design Speeds Transactions In Atos Worldline Hardware Security Module. Available at: http://www.electronicspecifier.com/design-automation/adyton-barco-silex-ip-atos-worldline-fpga-design-speeds-transactions-hardware-security-module, accessed on 9 July 2014.
[43] Shay Gueron. Intel Advanced Encryption Standard (AES) Instructions Set. Available at: https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/, accessed on 9 July 2014.

Appendix A: C MT6D Code

The source code of C MT6D developed as part of this research is stored in the git repository of the Virginia Tech Information Technology Security Office and can be accessed at the following location:

https://git.cirt.vt.edu/mt6d-c.git

Appendix B: Build Environment

This Appendix documents the build environment, compiler settings, and library versions used to compile C MT6D and the Python packages used by Python MT6D for the performance tests detailed in Chapters 4 and 5.

The hardware used in the tests is the GuruPlug Server [34], manufactured by GlobalScale Technologies. The GuruPlug Server is used for the two host devices and the GuruPlug Server Plus (which has two Ethernet ports) is used for the gateway devices. The devices run the Angstrom Distribution, version 2011.09, with Linux kernel 2.6.37.6 and are built using the OpenEmbedded build framework [35]. The GNU Compiler Collection (GCC) compiler system, version 4.5.3, was used to cross compile C MT6D for the GuruPlug target. The exact compiler command follows:

```
arm-angstrom-linux-gnueabi-gcc -march=armv5te -mtune=arm926ej-s -mthumb-interwork
    -mno-thumb -Wl,-O1 -Wl,--hash-style=gnu -g -ggdb -O0 -fno-omit-frame-pointer -pg
    -lcrypto -lssl -lnetfilter_queue -lip6tc -lxtables -pthread -o mt6d mt6d.c rehash.c stream.c
    icmp_packet.c icmp.c packet.c network.c -lnetlink session.c lib.c
```

The following packages were used to satisfy the library requirements of C MT6D:

libcrypto:	openssl-1.0.0e
libssl:	openssl-1.0.0e
libnetfilter_queue:	libnetfilter_queue-0.0.16
libip6tc:	iptables-1.4.9.1
libxtables:	iptables-1.4.9.1
libnetlink:	iproute2-2.6.38

The following OpenEmbedded Python packages were used by Python MT6D:

python-2.6.6	python-pycairo-1.8.0
python-dbus-0.83.2	python-pygobject-1_2.20.0
python-ipy-0.75	python-pygtk-2.16.0
python-libpcap-0.6.4	

python-pycrypto-2.0.1+gitrd087280d7e9643a3e3f68f209932119fe6738b3c