Improving the Security, Privacy, and Anonymity of a Client-Server Network through the Application of a Moving Target Defense

Christopher Frank Morrell

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

Doctor of Philosophy
in
Computer Engineering

Joseph G. Tront, Chair
Randolph C. Marchany
Scott F. Midkiff
Thomas C. Clancy
Danfeng Yao

March 17, 2016
Blacksburg, Virginia

Keywords: IPv6, Security, Privacy, Moving Target Defense, Client Server Network
Copyright 2016, Christopher F. Morrell

THIS PAGE INTENTIONALLY LEFT BLANK.

Improving the Security, Privacy, and Anonymity of a Client-Server Network through the Application of a Moving Target Defense

Christopher F. Morrell

ABSTRACT

The amount of data that is shared on the Internet is growing at an alarming rate. Current estimates state that approximately 2.5 exabytes of data were generated every day in 2012. This rate is only growing as people continue to increase their on-line presence. As the amount of data grows, so too do the number of people who are attempting to gain access to the data. Attackers try many methods to gain access to information, including a number of attacks that occur at the network layer.

A network-based moving target defense is a technique that obfuscates the location of a machine on the Internet by arbitrarily changing its IP address periodically. MT6D is one of these techniques that leverages the size of the IPv6 address space to make it statistically impossible for an attacker to find a specific target machine. MT6D was designed with a number of limitations that include manually generated static configurations and support for only peer to peer networks. This work presents extensions to MT6D that provide dynamically generated configurations, a secure and dynamic means of exchanging configurations, and with these new features, an ability to function as a server supporting a large number of clients.

This work makes three primary contributions to the field of network-based moving target defense systems. First, it provides a means to exchange arbitrary information in a way that provides network anonymity, authentication, and security. Second, it demonstrates a technique that gives MT6D the capability to exchange configuration information by only sharing public keys. Finally, it introduces a session establishment protocol that clients can use to establish concurrent connections with an MT6D server.

THIS PAGE INTENTIONALLY LEFT BLANK.

Improving the Security, Privacy, and Anonymity of a Client-Server Network through the Application of a Moving Target Defense

Christopher F. Morrell

GENERAL AUDIENCE ABSTRACT

Ensuring the security of computers on the Internet continues to grow in importance as more of our lives exist on-line. Traditionally, security experts add layers of defense to a computer or network in order to prevent malicious users from gaining access to the data held within that computer or network. No defense mechanism is perfect, and malicious users will continue to attempt to bypass any defenses until they are successful.

Computers on the Internet maintain an address at which other computers can send messages. Since the advent of the Internet, these addresses have been 32 bits long, which provides approximately 4 billion addresses. As more and more computers have been added to the Internet, the number of available addresses has decreased, resulting in a recent push to move from the 32-bit addressing scheme to a 128-bit addressing scheme, referred to as Internet Protocol version 6 (IPv6). Utilizing 128 bits to represent an address results in more addresses available than there have been nanoseconds since the Big Bang.

Previous research has demonstrated a technique that involves changing the addresses of two computers every few seconds, while continuing to maintain active conversations between the computers. Imagine being able to change your mailing address several times a day while still maintaining the ability to send and receive mail. This method, called Moving Target IPv6 Defense (MT6D), relies on the massive number of addresses available in IPv6 to make it practically impossible for some adversary to locate a specific computer that they wish to attack.

The work presented in this dissertation provides an extension to MT6D in order to support client/server networks. Each time a user requests a webpage or watches

a video, a conversation is created between the user's client computer and some other computer acting as a server. In a client/server network, there are generally a large number of clients that are all supported by a single server or small group of servers. In the described scenario, the user's computer is referred to as the client, while the machine providing the webpage or video is referred to as the server. MT6D was originally designed to support conversations that only occurred between two user's computers, not between a large number of clients and a server.

Additionally, MT6D originally required a great deal of pre-coordination and configuration to be conducted by users before they could establish a connection between their computers. This pre-coordination and configuration is a factor that greatly limits the number of users that could communicate concurrently due to the logistics required to distribute and apply configuration data. In support of extending MT6D, this research also presents a scheme that uses a form of anonymous escrow through which a server can securely share configuration information with any clients that may wish to establish a connection. By using this escrow service, client computers are able to establish a secure MT6D connection to a server with minimal pre-coordination and no manual configuration.

This work provides an increased level of privacy, security, and anonymity for computers that connect via the client/server model. These benefits come with a slight impact to a computer's performance on the network, although the negative impacts would not be notable to users except in extreme circumstances. The techniques presented here are intended to augment rather than replace the current security methods applied to a computer.

Dedication

The work presented here is dedicated to my family. Without the continued support of my wife, Leslie, and my children, I would not have been successful in this venture. Thank you for your understanding when the nights in front of the computer went longer than expected and when I was pre-occupied with work. I am especially grateful for my wife for enduring this experience, many others that have preceded it, and many others that will follow.

THIS PAGE INTENTIONALLY LEFT BLANK.

Acknowledgments

While this dissertation focuses on the work that I completed, it would not have been possible without the contributions and help from many people around me.

A great deal of respect and thanks goes to my advisor, Dr. Joseph Tront. Dr. Tront has been the advisor that every graduate student wishes to have. He was able to find the perfect balance of pushing me in the right direction and remaining hands-off. His approach allowed me the flexibility to explore research directions that may not have been directly applicable, but still contributed to the completion of my research.

I also thank Professor Randy Marchany for everything he contributed to my continued academic growth. Randy provided a sounding board for ideas as I progressed through my research. He was always quick to lend advice and keep me moving forward and never hesitated to provide direction when I went off track. He has the ability to provide levity when necessary, always ensuring that everyone working in the lab maintains perspective and a good sense of humor as they pursue their research. In particular, I appreciate the office space that I was able to occupy for my duration at Virginia Tech. Having a place to go to each day certainly helped to improve my efficiency.

To the rest of my committee, Dr. Scott Midkiff, Dr. Charles Clancy, and Dr. Danfeng Yao, thank you for your guidance and support throughout this process. I believe I picked the five busiest people at Virginia Tech for my committee, so I appreciate the time that each of you sacrificed for me.

To Dr. Scot Ransbottom and Dr. Dave Raymond, thank you both for your support throughout my academic and Army careers. In particular, I appreciate the

impact that Dr. Ransbottom has had on me both as an academic and an Army officer since we first met when I was a cadet at West Point. It was his advice in the fall of 1999 that started me down the path that has ultimately led to my current position.

To Dr. Matt Dunlop, without the effort that you put in to the development of MT6D, my ideas would have gone nowhere. Your hard work provided the foundation upon which I was able to build, culminating in the research that is presented in this document.

To my fellow IT Security Lab graduate students, Stephen Groat, Reese Moore, Phil Kobezak, Matt Sherburne, Dileep Basam, Kimberly Zeitz, Mike Cantrell, and Mark DeYoung, thank you for all of your help getting to where I am today. It was through our friendships and many hours of conversations that this work came to fruition. In particular, I appreciate the specific research contributions of Reese Moore and Mike Cantrell. Reese helped to develop the DHT Blind Rendezvous idea without which this work could not have happened. Mike provided a great deal of assistance through the implementation of both the MT6D server and the server demonstration website. The hours of assistance contributed by both of them contributed greatly to my ability to complete this research.

To the undergrads of the IT Security Lab, Andrew Heatwole, Franki Yeung, Peter Cho, and Alex Hsu, thank you for your contribution to this research. The effort provided by each of you helped to round out my research, but allowed me to remain focused on the central topics. I learned a great deal from each of you and hope that you are able to take some knowledge away that you learned through our interactions.

Finally, to the United States Army and the department of Electrical Engineering and Computer Science at the United States Military Academy, thank you for providing me with the opportunity to pursue a doctorate degree.

Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK.

List of Algorithms

THIS PAGE INTENTIONALLY LEFT BLANK.

List of Code Samples

THIS PAGE INTENTIONALLY LEFT BLANK.

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 1

Introduction

Traditional computer security treats machines like castles in an effort to secure them. Security experts build layers upon layers of security in a manner akin to walls and moats, creating what is known as a defense in depth. This layered defense technique works to some degree, but every castle has weaknesses. What if a castle, including all of its layers of security, could move to arbitrary locations at will? An adversary would be required to greatly increase their commitment to reconnaissance resources in the effort to find their target before they could ever begin to conduct an attack. Of course this functionality would only work if the people who draw services from the castle knew where it moved to, so that their service remained uninterrupted. This analogy very closely describes a network-based moving target defense where the castle is a computer, and the space that it moves within is the local subnet's logical address space. This specific technique will not work well in a legacy IPv4 network, since address space within a subnet is often limited to only 256 addresses. However, as the Internet continues to move towards IPv6, the ideas presented here show a great deal of promise in the improvement of security, privacy, and anonymity on the network.

The Internet's transition from Internet Protocol version 4 (IPv4) to Internet Protocol version 6 (IPv6) will begin to quicken in pace due to the fact that the address space available within IPv4 is quickly running out [54]. In fact, as of early 2016,

only the African Regional Internet Registry (AFRINIC) has not begun limiting distribution of IPv4 addresses based on address space exhaustion. IPv6 solves the IPv4 address exhaustion problem by moving from a 32-bit address space to a 128-bit address space. The IPv6 address space contains 2^{128} addresses or significantly more addresses than there have been nanoseconds since the Big Bang. The international community is slowly moving towards implementing IPv6 across the Internet, which brings with it a number of opportunities to improve the security, privacy, and anonymity of users on the network.

One technique that attempts to improve the security, privacy, and anonymity on the Internet is a network layer moving target defense. A moving target defense utilizes the immense space within IPv6 in order to move machines arbitrarily throughout the address space in order to avoid detection by malicious actors. As machines are logically moved around the network, a bad actor must commit their resources to finding the moving machine as opposed to attempting to bypass or break through any security defenses installed on the machine.

Moving Target IPv6 Defense (MT6D) is a specific implementation of a network layer moving target defense that was introduced by researchers at Virginia Tech [33]. MT6D is a proof of concept implementation that allows two nodes to utilize a network layer moving target defense while continuing to maintain open Transport Control Protocol (TCP) sessions. Two severe limitations of the original implementation of MT6D are that the researchers focused solely on peer-to-peer networks and provided no means to dynamically exchange required session establishment configuration data.

This research proposes modifications to MT6D that overcome these shortcomings in order to provide support for the more commonly occurring client/server networks. This research also provides a facility through which configuration data can be exchanged at scale and securely. Section 1.1 outlines the problem statement that this research intends to address. Section 1.2 provides some background and motivation for this research. Objectives of this research are described in Section 1.4. Section 1.5 provides detailed research questions. Section 1.6 describes the methods used to conduct this research, while Section 1.7 will provide the structure for the rest of this document.

1.1 Problem Statement

The purpose of this research effort is to design, implement, and test a technique that extends a network layer moving target defense to support a client/server network. The moving target defense research community is small, and to this point no one has attempted to apply the techniques to anything larger than a peer-to-peer network. This research attempts to find a way to exchange configuration information securely and dynamically in order to facilitate a large scale deployment of a moving target defense. Additionally, this research works to discover the limits of client to server ratio. Finally, this research attempts to push those limits by improving the efficiency and performance of the moving target defense server.

1.2 Background and Motivation

The vast majority of services that users consume on the Internet today are client/server-based services. For example, every visit to a webpage and every email sent and received involves the interaction of the user's computer (the client) with some other computer that is providing a service (the server). In order to provide a service, the server must remain accessible to the clients that wish to connect to it. In many cases, servers also contain a great deal of important information related to each of their clients such as usernames, passwords, or financial information. It is for these reasons that much of the security research done today focuses on protecting servers, the communication that they conduct, and the data which resides in them.

Network layer moving target defense is an interesting idea that shows promise as a means to augment the anonymity, security, and privacy of a computer on the Internet. Unfortunately, all of the current network moving target defense work focuses on peer-to-peer connections. Examples of this peer to peer focus are apparent in the ideas presented by MacFarland [51], Antonatos [12], and Chavez [23]. In particular, the MT6D scheme introduced earlier in this chapter provides an implementation of a network-based moving target defense which focuses on peer-to-peer communication

and conducts address changing at the host level rather than the network level.

Sherburne [73] and Preiss [69] consider the application of a moving target defense to Internet of Things (IoT) devices. Within the IoT, a large number of low power sensor devices must send data to some point for aggregation. This case is an example of needing to extend the ideas behind a network-based moving target defense in the direction of supporting a client/server network.

The work presented here overcomes many of the issues present in the original design of MT6D by using a Distributed Hash Table (DHT) as a means to conduct a blind rendezvous, which allows the secure exchange of MT6D configuration data. Primarily though, this work presents the design, implementation, and testing of modifications to the MT6D scheme which allow a single server to provide services to some large number of clients.

MT6D is not a single solution to security and privacy on the Internet, but does aim to solve a number of specific problems and defend against a specific set of attacks using its ever-changing addresses. Potential attacks include, but are not limited to, address-based Denial-of-Service, replay, session hijacking, and man-in-the-middle attacks. These targeted attacks rely on the fact that a host traditionally remains at a fixed location in the network. As hosts change their addresses, the attacker is left behind without the ability to find the new address that the MT6D hosts have moved to. While MT6D does serve as a prevention mechanism for many of the network layer attacks, it does not prevent application layer attacks.

The implementation of this MT6D server, and its ability to exchange configurations securely and dynamically, provides additional protections that the original implementation of MT6D failed to provide. The movement from statically to dynamically generated configurations ensures that the risk of compromised passwords and replay attacks are removed as issues. Additionally, the ability to distribute configuration files through the DHT eliminates the risk of storing static configurations locally and provides the capability to distribute to a much larger audience without the logistical overhead of manual configurations. Finally, the added dynamism presented in this paper provides a capability that could be leveraged to enable moving target defense configuration changes as the security level of the network fluctuates.

1.3 Identifying the Threat

In this work, the threat is persistent and is conducting attacks targeted against specific hosts. These targeted attacks may be either active or passive in nature. Active attacks are those in which the adversary desires to disrupt, intercept, or modify the network communications of a host. These attacks require that the adversary have knowledge of the network infrastructure and more importantly the logical addresses of hosts on the network. Passive attacks are those that involve the attacker silently listening and waiting for information to arrive. The attacker requires enough contextual information to be able to consolidate packets into flows so that data may be extracted.

In security research a Trusted Computing Base (TCB) is defined as the set of computers or systems that are critical to performance, must be protected, and hence are assumed to be trustworthy. Computers or systems that exist outside of the TCB are those that an adversary could gain control of, but would not compromise the integrity of the overall scheme. In this research, the TCB includes the machine on which the MT6D software is running and the local subnet in which the MT6D hosts are located. Since this work focuses on network protocols, any vulnerabilities on the machine running the server are outside of the scope of this research. The local subnet is included in the TCB due to the fact that MT6D provides network layer (OSI layer 3) anonymity. Due to the nature of TCP/IP and Ethernet networking, anonymity at layer 3 does not provide any protections against traffic sniffing on the local subnet since every Ethernet frame transmitted from the MT6D host contains that host's Media Access Control (MAC) address and therefore is easily traceable back to its origin. If an adversary is able to listen to network traffic inside of the subnet on which an MT6D host is located, the anonymity and privacy that is provided by MT6D will be compromised, although the data remains encrypted and secure. Should an adversary gain access to a client that is participating in an MT6D conversation, that client and its data may be compromised, but this compromise alone does not imply that the server is also compromised.

1.4 Research Objectives

The primary objective of this research is to design, implement, and test modifications to the MT6D scheme in order to allow one of the nodes to serve as a server for a large number of clients. In order to achieve this objective, the research must provide some means to securely distribute configuration information to a large number of clients. Additionally, this work must demonstrate a functional MT6D server that does not reduce any of the capabilities provided in the original implementation of MT6D. Finally, this research must determine the maximum number of clients that can be served by a single server. By achieving these objectives, the security, privacy, and anonymity of a server will be improved while MT6D will gain a great deal of functionality.

1.5 Research Questions

This research addresses the following questions.

1. How can MT6D be modified in order to support a client/server network?
2. How can a large number of clients be configured efficiently and securely?
   (a) What is the minimal amount of information that must be pre-shared between MT6D nodes?
   (b) Is there a solution that will allow nodes to anonymously and securely exchange configuration information?
   (c) How can information be authenticated?
3. How many clients can the server support?
   (a) What is the baseline for the maximum number of clients?
   (b) What efficiencies can be introduced to raise that number?
   (c) What is the impact on the network of supporting a large number of clients?
   (d) How many clients should share a single MT6D interface on the server?

1.6 Methodology Overview

This research was conducted in three primary phases which include theoretical, simulation-based, and real-world implementation. The first phase focused on the distribution of configuration data in an anonymous but authenticated and secure manner. The second phase focused on determining the limits on the number of IPv6 addresses that can be bound to a machine while continuing to function. This phase also included an effort to improve those limits as well as examining the impacts of MT6D on the network. Finally, phase 3 consisted of final design, building, and testing of an MT6D server.

1.7 Organization of Dissertation

The following provides a brief overview of how the rest of this document is organized. Chapter 2 provides an overview of works that are related to this research from which the inspiration for this work was derived. Chapter 3 describes the multiple protocols and tools that are used through out this research. Chapter 4 introduces a blind rendezvous technique that allows two nodes on a network to exchange small pieces of information through the use of the BitTorrent DHT. Chapter 5 provides an MT6D focused implementation of the DHT blind rendezvous scheme that is presented in Chapter 4. Chapter 6 describes the design of the MT6D server and details the modifications to the MT6D scheme that are required in order to support a client/server network. Chapter 6 also provides the implementation details and analysis of the MT6D server. Chapter 7 focuses on the problems associated with supporting a large number of clients with a single MT6D server in addition to the impacts that an MT6D server has on the network. Chapter 8 describes a visualization tool that was developed in order to better see what is happening in MT6D. Finally, Chapter 9 provides thoughts on the future of this research and concludes the document.

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 2

Literature Review

This research is based on a large body of work from a variety of sources. At the heart of the work presented here is MT6D and the derivative works also based on the original MT6D scheme. Additionally, there are a number of different approaches to solving the network-based moving target defense problem, although they focus very heavily on peer to peer models or require modifications to network architecture.

2.1 Moving Target IPv6 Defense

MT6D was originally presented by Dunlop, et al. [33, 34], researchers at Virginia Tech. It was designed to solve the weaknesses inherent in the Extended Unique Identifier-64 (EUI-64) address calculation that is part of the Stateless Address Autoconfiguration (SLAAC) protocol and was explored by Groat, et al. [40]. The primary weakness in SLAAC is the fact that the last 64 bits of the IPv6 address, the Interface Identifier (IID), do not change even as a computer moves around the Internet. This unchanging portion of the address means that a malicious actor could follow a machine around the network, thus decreasing the privacy of the user. MT6D solves this problem by eliminating the use of the SLAAC generated address and provides a facility through which communicating peers can change the IID portion of their IPv6 address every few seconds. IPv6, SLAAC, and MT6D will be discussed in much

greater detail in Chapter 3. MT6D is the basis for many of the methods required to design and implement a network-based moving target defense server.

2.1.1 MT6D and the Internet of Things

In his Master's degree thesis, Sherburne [74] describes a means to modify MT6D to support address hopping in very low power and resource constrained devices. Additionally presented by Sherburne [73] and Preiss [69], the authors present a technique that relies on changing the MAC address in order to force the operating system to change the generated IPv6 address. Unfortunately, the authors were not able to push their research far enough to the point that they could conduct the hashing required to calculate MT6D addresses, but they do succeed at providing inspiration for this work.

IoT devices require some central collection or control point to which all of the data that they collect is dispatched. This service is normally provided by some type of server that can collect, consolidate and perform some processing on the data in order to present it to the user in a meaningful manner. By pushing these low powered IoT devices into MT6D, the authors require that some server be designed that is compatible both with MT6D, but also can run some software that performs the collection and processing of the IoT data.

2.1.2 MT6D Honeypot

Basam, et al. [14, 15] explores a system that serves as a monitor and honeypot for machines that rely on MT6D for privacy and security. Through the use of his technique, a user of MT6D could monitor IPv6 addresses that had been previously assigned by MT6D, but were no longer in use. By monitoring these previously used addresses, one could gain some insight into the unpredictability of the MT6D address generation scheme. If the address monitor sees a large number of MT6D addresses being used by unexpected hosts, there is likely either a weakness in the scheme or keying material has been compromised. By connecting this monitor to an

MT6D host, a keying change could be triggered. This need to dynamically generate configuration data is not something that was supported in the original design of MT6D, but helped to inspire the addition of this capability in the work presented here.

2.2 Alternatives to MT6D

In 2013, a group of researchers at MIT Lincoln Laboratory conduced a *Survey of Cyber Moving Targets*, which included an entire section dedicated to network-based moving target defense schemes [65]. The common theme among every technique presented in the paper is that researchers focus on putting the address changing capability at the network level where MT6D pushes the capability to the host level. Additionally, the work reviewed in this paper only explore moving target connections between peers as opposed to working towards some form of moving target defense server.

2.2.1 Moving Target Theoretical Work

Zhuang, et al. [86], present a theoretical model to which they believe all moving target systems should be applied. Most importantly, this work provides a common set of terms and definitions that are applicable across the entire body of moving target defense work. Key thoughts from this paper include the definition of a moving target system as one in which there is some adaptation to a system from within its configurable space that results in a valid but different configuration than before the adaptation. The authors also emphasize the importance of randomization in any moving target scheme, thus ensuring that some adversary is not able to predict adaptations before they happen. MT6D abides by this theory by randomly selecting IPv6 addresses from the pool of valid addresses, always ensuring that each newly created address is different from the previous.

2.2.2 Network Translation Moving Target

One of the primary techniques that researchers leverage in the design of moving target defense systems is to modify the network. This technique approaches the problem from a different direction than MT6D, where modifications are conducted on the host. Some of the earliest work in this field was conducted at Sandia National Laboratories in 2002 [53]. The authors do not provide a specific scheme that would be used, but rather a high level description of several techniques that could be used to obfuscate communication between two peers. A more specific technique was presented by researchers at University of Southern California in which they build a number of overlay networks that could be used to route traffic in different ways [79]. By using these overlay networks, the researchers argue that they could quickly re-route traffic to its desired endpoint without needing to modify the underlying network topology.

Software Defined Networking (SDN) is quickly gaining popularity in the networking community, with it's growing popularity a flurry of SDN research has resulted in a number of different network-based moving target defense schemes, including MacFarland [51], Jafarian [46], Chavez [23]. SDN is a technology which moves away from static network topologies that were controlled purely by hardware switches and routers and towards a more intelligent network that can be controlled and modified more readily through software. Authors present different methods through which an SDN controller can perform some type of network address translation at the network boundary in each of these papers. Through the use of this technique, public facing properties of network hosts can be modified arbitrarily without actually changing them on the host. The SDN controller simply re-writes the modified information in the packets before sending them into or out of the network. These techniques show promise, but require the use of software defined networking and control of the device that resides on the network boundary. This limitation forces reliance on network owner/operator cooperation in order to successfully hide a host.

Still other techniques, such as those presented by Yackoski [81] and Al-Shaer [11] suggest that network-based moving target defense should be controlled by some cen-

tral device that has full awareness of the state of the network and provides configuration modification guidance to hosts through some out-of-band channel. The techniques presented in these papers also show a great deal of promise, but sacrifice some of the benefits of using a network-based moving target defense. Depending on how the out-of-band communication is conducted, it is possible that the privacy and anonymity provided by moving target defenses could be weakened. Work presented by Zhang, et al. [84] argues that out-of-band communication is impractical and potentially unreliable. The authors do offer an alternative technique that utilizes in-band communication for the distribution of network configuration changes which could be leveraged to improve some of the centrally controlled moving target defense systems.

2.2.3 Other techniques

Mobile IPv6 is a protocol defined in Request For Comments (RFC) 3775 [48] and RFC 3776 [13] that gives IPv6 addressed nodes on the network the ability to quickly modify their addresses without losing connection to communication peers. Researchers present a scheme that uses features provided by mobile IPv6 in order to build a network-based moving target defense system that is similar in design to MT6D [43]. The authors use virtual IPv6 addresses for communication with peers while maintaining SLAAC generated or statically defined IPv6 addresses on their hosts. By assigning hosts a large number of virtual addresses, peers are able to select an address from the assigned addresses at random for a specific packet.

Antonatos, et al. [12], present a technique described as Network Address Space Randomization (NASR). NASR is essentially a modification to the Dynamic Host Configuration Protocol (DHCP) in which hosts are forced to release their DHCP leases at arbitrary times. The authors argue that they are able to change not only the IP addresses of their hosts, but also a number of other network properties such as network gateways and Domain Name System (DNS) servers by controlling the DHCP server and the configuration information that is distributed. This technique requires a non-standard implementation of DHCP in which lease times are artificially

short and potentially changing in duration. According to DHCP updates presented in RFC 1541 [32], there is no minimum lease time. Therefore, the NASR address change interval would only be limited by the capacity of the network infrastructure and the DHCP server.

Privacy extensions are an Internet Engineering Task Force (IETF) approved protocol that was described in RFC 3041 [58] and updated in RFC 4941 [59] which provides a means through which SLAAC generated IPv6 addresses could by replaced by cryptographically generated addresses similar to the techniques described in MT6D. Privacy extensions is an attempt at improving the privacy weaknesses that are present in SLAAC; namely that the IID portion of the IPv6 address never changes. This issue is resolved with privacy extensions by forcing address changes daily.

2.3 Assessing Moving Target Defenses

In 2015, there were at least two separate papers that were published that focused on evaluating the effectiveness of and characterizing moving target defenses. The first focused on network-based moving target defense schemes [39], while the second took a much more generalized approach to evaluating research [82]. Both of these papers seek to expand on the MIT Lincoln Lab survey paper described in Section 2.2 by providing frameworks against which moving target defense technology can be rated. The first paper provides three specific properties that a functional network-based moving target defense scheme should provide. These are a moving property, an access control property, and a distinguishability property. The moving property states that a machine must change its location on the network to an unpredictable location periodically. The access control property states that only an authorized peer should be able to find a peer that is utilizing the moving target scheme. Finally, the distinguishability property states that the system should be capable of determining trustworthy clients from untrustworthy clients. The authors apply these properties as a means of assessing the effectiveness of several network-based moving target defense

systems, including an SDN based scheme and MT6D. Their results demonstrate that both schemes meet their minimum properties of an effective moving target defense, although the authors note two possible weaknesses in the original design of MT6D. They raise concerns regarding the possibility that MT6D could exhaust network resources and address an issue regarding multiple MT6D pairs using common keys. The work presented in this dissertation addresses both of those concerns and demonstrates that their concerns have been resolved.

2.4 Impacts of a Network-Based Moving Target Defense

A great deal of work has been contributed to making moving target defense systems strong in order to ensure that the systems actual improve the privacy and security of hosts on the network. Each time security functionality is added to a system, performance is impacted. Zhuang, et al., have spent a great deal of time studying the impacts of network-based moving target defense systems on the surrounding network. The authors provide the results of their analysis in addition to models and suggestions for assessing moving target defense impacts in three separate papers [85, 87, 88]. The first of these papers suggests techniques that leverage computer simulation techniques in order to asses the effectiveness of moving target defense systems. The authors describe a moving target based system that relies on a central controller and out-of-band configuration changes and build a network security simulation in order to measure the effectiveness of their scheme. The second paper extends the model described in their first paper into a theoretical model that they again use to asses the strength of their moving target defense system. In their third paper, the authors move away from assessing the security provided by their scheme, and rather focus on the impact to the network in which their moving target scheme exists. As in earlier work, the authors provide a theoretical model that can be used to assess the impact of a general moving target defense scheme on a network.

2.5 Differing Perspectives on Moving Target Defense

All of the work discussed in the previous sections focuses on the modification of some network properties in order to provide a level of privacy and security. A moving target defense scheme can keep a host from being discovered on a network by simply changing an IP address or a port number, and can therefore improve its security posture. This section presents research areas that are not considered moving target defense technology, but provide some inspiration to the work presented here.

2.5.1 Botnet Command and Control

A botnet is a collection of controlled computers (bots) across the Internet that are used for a variety of malicious purposes. In particular, botnets are the most common means for conducting Distributed Denial of Service (DDoS) attacks, but can also be used by phishers to maintain a large number of servers. Bots are created by installing some controlling software on a machine, generally through a trojan horse virus, phishing, or some other malware. There are a number of techniques and protocols that can be used to control the bots, but finding the members of the botnet on the Internet is a significant challenge. At the same time, botnet operators must attempt to keep the locations of their bots hidden so that authorities are not able to track them and disassemble their botnets. Add to this challenge the fact that bots are dispersed across the Internet.

DNS is essentially the Internet's yellow pages through which a Uniform Resource Locator (URL) is converted to an IP address, and is the reason that a user can type a URL into their browser and begin the process of requesting data from some other machine on the network. When used as designed, name to IP mappings within the DNS do not change often, helping to increase the stability of the network.

Botnet operators use a technique termed "fast-flux DNS," where nodes in the network register and unregister their addresses from the DNS at very short intervals.

By changing DNS entries every few minutes, botnet operators ensure accessibility even as machines leave the botnet [10]. At a high level, the quick movement from machine to machine looks similar to a moving target defense system.

While fast-flux DNS works for botnet operators, it does result in the creation of a great deal of DNS traffic. Researchers are working towards solutions that exploit this spike in traffic in order to discover and shutdown botnets [24]. In addition to discovering botnets through DNS traffic observation, researchers are also working towards the creation of tools that can identify botnets by monitoring a variety of command and control protocols [41].

2.5.2 Frequency Hopping Radios and Blind Rendezvous

Frequency hopping is a method used by radios to prevent the jamming of radio frequencies [35, 76]. In frequency hopping, radios quickly move between an approved set of frequencies within a band. This idea of changing frequencies in order to prevent jamming is very similar to network-based moving target defense schemes in that they share a goal of changing some specific property of their communication scheme to avoid some negative action from an adversary. An ongoing area of research in the field of frequency hopping and cognitive radios is solving the problem of conducting a blind rendezvous [36, 42, 72]. In cognitive radio, a blind rendezvous is the act of ensuring that two radios are on the same frequency at the same time in order to establish a connection. There are a number of techniques that are used to execute this blind rendezvous [66], but all focus on the fact that the number of available frequencies is small. Some of these techniques were examined to provide inspiration for conducting a blind rendezvous in MT6D, although the difference in search space is extreme.

2.6 Summary

This chapter has presented an overview of related work and inspiration for the research that will be presented throughout the rest of this dissertation. Primarily, the

research discussed here lays the foundation for developing an understanding of how the development of an MT6D server compares with similar technologies. The techniques that compare most closely with MT6D are those network translation moving target schemes that rely on modifying the underlying network architecture rather than the host. The approach that MT6D takes, and the approach that continues to be developed throughout this research, is the modification of addresses on the host rather than the network edge. While this technique does require modification of each host that wishes to use the scheme, it can be easily implemented in any network that supports IPv6, and does not require the support of network owner/operators.

Chapter 3

Supporting Protocols and Tools

3.1 Internet Protocol Version 6

IPv6 is the 128-bit replacement to the aging 32-bit IPv4 standard. IPv6 was originally published as part of RFC 2460 [29] in December of 1998 when it was realized that IPv4 would not be able to support the number of IP addresses that would be required as the Internet continued to grow. The biggest benefit provided by the move to IPv6 is the immense growth in the number of addresses available as addresses move from 32 to 128 bits in length. A 128-bit address space provides 340 undecillion or 340 billion billion billion billion addresses, while a 32-bit address space provides only 4 billion. Since the human mind has no way to comprehend this number, analogies attempt to make these massive numbers more understandable.

Imagine that an IP address has volume and the entire 32-bit address space available in IPv4 is fit inside of a ball with a 1-meter diameter. At that size, each cubic millimeter of space holds 8 addresses. Imagine now that an IPv6 address has the same volume. In order to hold all of the addresses available in the 128-bit IPv6 address space, a ball with a diameter of more than 4 million kilometers, which is more than 3 times the diameter of the sun, would be required. It would take light more than 14 seconds to travel the diameter of this container. While still difficult to fully comprehend, this extreme analogy communicates an understanding of the

massive increase in capacity provided by only 96 extra bits.

In IPv6, the address is generally split into three pieces which each represent different parts or layers of the network. The first 48 bits are used to represent the routing prefix, the next 16 bits represent the subnet identification, and the last 64 bits, also known as the IID, represent the host. These three portions are used together in order to route a packet to a particular network on the Internet, a subnetwork within that larger network, and a specific host within that subnet. For the matters of this work, it will be easier to simply separate the address into two pieces. As shown in Figure 3.1, the first 64 bits will deal with routing and will be referred to as the network portion. The second 64 bits will still represent the host on the subnet.

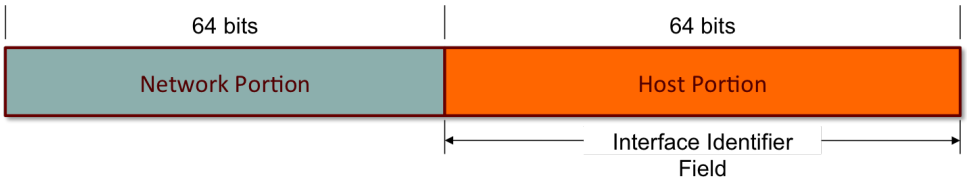

Figure 3.1: A simplified representation of an IPv6 address that is broken into only two portions.

On a 64-bit subnet, there are approximately 2^{64} or $1.845 \cdot 10^{19}$ distinct IIDs available. In stark contrast, there are only a total of 2^{32} or approximately 4 billion addresses available in the entirety of the IPv4 address space. Due to the limited address space in IPv4, assigning addresses requires a great deal of planning and close control from network administrators in order to ensure efficient usage. In fact, the IPv4 addressed Internet relies very heavily on RFC 1918 [71] private addresses in order to function. These private addresses permit a large number of addresses to masquerade as a single address. In contrast, it is completely feasible in IPv6 to permit a client to simply select their own IID, since the probability of a collision is almost zero. According to the birthday paradox, it would take approximately $2^{64/2}$ or just over 4 billion attempts to have a greater than 50% likelihood of two addresses colliding inside of a 64-bit IPv6 subnet. This fact permits the majority of IPv6 networks to rely on IPv6 SLAAC in order to assign addresses to their clients and

is also a key feature that is leveraged in order to make network level moving target defense schemes function.

3.2 IPv6 Stateless Address Autoconfiguration

IPv6 SLAAC is an automatic addressing scheme that was formalized in RFC 2462 [77] and updated in RFC 4862 [78] that provides a machine with the ability to generate its own an IPv6 address. Hosts are permitted to generate their own IID's by using the EUI-64 format [44]. EUI-64 functions by splitting the 48-bit MAC address in half and inserting a 16-bit hex value of `0xffee` between the halves of the MAC address. The host also sets bit 7 of the IID, which is the universal/local flag, to 1. In order to use a correct routing prefix and subnet id for their given network, the host must request this information from their local router by using Neighbor Discover Protocol (NDP) [60].

NDP is the IPv6 replacement for IPv4's Address Resolution Protocol (ARP). It consists of five messages that are used by routers and hosts in order to determine which other hosts are on the network around them. In SLAAC, a host sends a router solicitation message onto the network. The router will hear this request and respond with a router advertisement message which contains the first 64-bits of the IPv6 address that the host will use.

The host now concatenates the router provided network information with the EUI-64 generated IID resulting in a unique 128-bit IPv6 address. Since most machines on the Internet use this same method each time they generate an address, it is likely that the IID will be the same for a given interface on a machine. It is also worth remembering that bits 7 and 25-40 of the IID will be the same for every SLAAC generated address, no matter the machine generating the address.

3.3 Moving Target IPv6 Defense

The MT6D scheme was originally proposed by Dunlop, et al. in the 2011 paper, *MT6D: A Moving Target IPv6 Defense* [33]. MT6D is a network layer moving target defense scheme that leverages the enormity of the IPv6 address space and the fact that nodes typically self-assign addresses within a subnet to allow nodes to rapidly and repeatedly change their network address without disrupting ongoing network connections. As with IPv6 SLAAC, the fact that the typical IPv6 subnet is 64 bits, there is a near zero probability that MT6D generated addresses will collide with other nodes on the network. The probability of a collision on a subnet is calculated using Equation 3.1 where h is the number of active hosts on a 64-bit IPv6 subnet.

$$P_c = \frac{h}{2^{64}} \tag{3.1}$$

Through the use of Equation 3.1, it is easy to see that the probability of selecting a currently assigned address in a network with 100,000 occupied addresses is only $5.42 \times 10^{-15}\%$. In fact, even if the entirety of the IPv4 address space were active within a 64-bit IPv6 subnet, there would only be a $2.33 \times 10^{-10}\%$ chance of selecting a currently assigned address.

MT6D algorithmically generates addresses based on a node's SLAAC generated or statically assigned IID, a secret key, and the current time using the following equations:

$$IID'_{x(t_i)} = H(IID_x||K_s||t_i)_{0\rightarrow 63} \tag{3.2a}$$

$$Addr_{x(t_i)} = Subnet_x||IID'_{x(t_i)} \tag{3.2b}$$

The MT6D IID for time slot t_i is generated above in Equation 3.2a by using a cryptographically secure one-way function such as SHA256 to hash the concatenation of the base IID, a shared secret key K_s, and the time slot value. After hashing, the first 64 bits of the digest are used as the MT6D IID for the current time slot. The

MT6D address is then calculated by concatenating the machine's original IPv6 subnet with the newly calculated IID as in Equation 3.2b. The original subnet portion of the address must stay intact so that packets can be routed to the correct network. Nodes use the scheme to calculate addresses for both themselves and their communication peer node. Calculating a peer node's addresses ensures that MT6D packet headers can be addressed with the destination node's ever-changing MT6D generated IPv6 addresses. All configuration information is pre-configured statically and nodes must know their peer's subnet, IID and K_s. Finally, time must be synchronized between communicating peers, ensuring that MT6D IID addresses are calculated using the same time slot value, t_i.

The length of the address rotation time period is a network parameter which must be prearranged before the system goes into use. The shorter the time period, the more resources are utilized for binding and unbinding addresses but the more agile and resilient to attackers monitoring the network traffic. Conversely, a longer rotation period utilizes fewer resources, but creates a larger attack window. In order to increase the reliability of the network connection in the presence of higher latency and clock drift MT6D maintains three address: the previous window address, the current window address, and the next window address.

Many network firewalls will reject protocols which they do not recognize. Because of this, MT6D encapsulates traffic inside of a User Datagram Protocol (UDP) datagram. An outsider observing network traffic would see many different IPv6 nodes exchanging UDP datagrams; however, the applications running on the MT6D nodes will be exchanging packets with whatever arbitrary protocols are needed in the course of their communication with their peer. This encapsulation method allows unmodified applications to take advantage of the security benefits provided by the MT6D association while maintaining ongoing connections, oblivious to the changing network addresses.

In addition to calculating new IPv6 addresses, MT6D provided a method through which UDP port numbers can also be changed at each address rotation time period. While this port number rotation is not a required part of the MT6D scheme, it does provide an additional layer of obfuscation. If port numbers are allowed to stay the

same as IPv6 address change, it would be possible for some adversary to reconstruct network flows based solely on port numbers. To this end, MT6D adds port calcuation according to the following equations:

$$Src\_Port_{x(t_i)} = H(IID_{src}||K_s||t_i)_{64\rightarrow 79} \tag{3.3a}$$

$$Dest\_Port_{x(t_i)} = H(IID_{dest}||K_s||t_i)_{64\rightarrow 79} \tag{3.3b}$$

Port calculations simply rely on bits 64 through 79 of the standard MT6D address calculation equations. The source port comes from the source's address calculation results while the destination port comes from the destination's address calculation results.

The rapidly changing network addresses in MT6D allow a machine to logically move throughout a sub-network while maintaining connectivity with those "good" nodes that it wishes to communicate with and know where it is moving to. This logical movement presents a challenge for a would-be attacker, and forces a paradigm shift in the attacker's tactics. While traditional attack methods follow phases of reconnaissance, attack, exploit, and clean-up, MT6D forces the adversary to commit all of their resources to the reconnaissance phase. The enormity of the IPv6 address space, even within a single subnet, makes it incredibly unlikely that an adversary will be able to locate and continually communicate with the target machine simply through random chance. Additionally, MT6D is not intended to be deployed as the only line of defense, but instead as a single layer with a defense-in-depth strategy with other, more traditional, layers providing additional security to the protected machine.

3.4 Key Exchange and Agreement

One of the essential components of a distributed system is the ability to establish secure connections through the use of a shared key. There are many methods available to establish these keys including pre-sharing keys and executing a key exchange.

A pre-shared scenario requires that peers wishing to communicate agree on a key or exchange a key out-of-band before any secure communication is able to take place. In key exchange scenarios, peers are able to establish a shared secret key through the use of public key cryptography. An end-point uses its peer's public key to encrypt information that can only be decrypted by using the peer's private key. Other key exchange mechanisms make use of long-term signing keys to verify the identity of the sender and the integrity of the message, and use this tamper-proof channel to exchange portions of a key negotiation resulting in a shared secret. This ability to exchange secret information over an insecure public channel provides easy and secure distribution capabilities. While this exchange of information is secure, the encryption protocols used are relatively inefficient, and should not be used for bulk encryption. For this reason, a key exchange is used to simply establish a common secret key that can then be used in a more efficient bulk encryption scheme.

An example implementation of a key exchange that has some parallels to this work is the method through which Secure Socket Layer (SSL) [38] and Transport Layer Security (TLS) [31] establish a secure session. In SSL/TLS, two machines use some key exchange method (Diffie-Hellman, Rivest-Shamir-Adleman (RSA), Elliptic Curve Diffie-Hellman, or others) to establish a common secret key. This secret key is then used in some symmetric encryption scheme to encrypt and authenticate all data exchanged between the machines. This technique permits the machines to establish a secure connection with no pre-shared information.

3.5 BitTorrent Distributed Hash Table Protocol

BitTorrent is a peer-to-peer protocol that is used for file sharing wherein users distribute the bandwidth of the file transfer by uploading and downloading portions of a file to and from each other. Because of this design, a download with a large number of downloaders can be sustained without relying on the bandwidth of a single server [28].

The BitTorrent protocol was originally designed to include a server called a

"tracker" that would be used to coordinate peers. A BitTorrent client wishing to download a file would extract information about the file from a `.torrent` file which included a list of BitTorrent trackers. The client would then contact the trackers with information about itself, and receive a list of peers from which they could transfer that particular file [28].

The BitTorrent tracker represented a single point of failure for a BitTorrent cloud, and many high profile trackers have been targeted both by DDoS attacks and legal threats. In the interest of making the BitTorrent network more resilient against such attacks, and to further decentralize the network, a BitTorrent DHT protocol was introduced in 2008 [50]. With this modification to the BitTorrent protocol, the tracker was no longer a central point of failure for the BitTorrent cloud as each peer in the swarm becomes a tracker.

The BitTorrent DHT protocol is based on the Kademlia DHT presented by Maymounkov and Mazières [52]. Kademlia is a peer-to-peer DHT where node identities and data keys are both 160-bit opaque values. When a node wants to store a Key-Value pair in the DHT, they push the value to a node with an ID which is "close" to the data key based on an XOR metric. The Kademlia protocol allows nodes to efficiently learn about peers with addresses closer to the key at which they are attempting to store or retrieve data.

The BitTorrent DHT protocol [50] specifies several "DHT Queries" which nodes can use to walk through the Kademlia DHT and locate the peers currently downloading a specific torrent, and add themselves to the swarm so that other peers can locate them. The four queries currently specified and their use is given in Table 3.1. When a node wishes to locate a torrent, it calculates a hash of the desired file and executes a *get_peers* query in order to find nodes hosting the torrent in question. Once downloaded, a node sends an *announce_peer* query to notify the DHT that it is a possible source for the given torrent.

Table 3.1: Currently defined DHT queries for the BitTorrent DHT.

Query	Description
`ping`	A simple ping/pong to establish if a node is still alive.
`find_node`	Used to acquire the contact information (IP/port) for a node based on its node ID.
`get_peers`	Used to retrieve information about peers in the swarm for a specific torrent.
`announce_peer`	Used to add oneself to the list of peers in the swarm for a specific torrent.

3.6 Neighbor Discovery Protocol

With the introduction of IPv6, the NDP was created to replace the ARP of IPv4. Originally described in RFC 2461 [60] and updated in RFC 4861 [61], NDP provides a means through which IPv6 devices can communicate and track other IPv6 devices on the local subnet. These tasks are completed by translating between IPv6 and link-layer addresses, maintaining router information, and periodically updating the reachability status of other devices on the subnet through the use of Internet Control Message Protocol version 6 (ICMPv6) messages. NDP standardizes many of the external capabilities of ARP such as the ability to label routers, eliminating the need for many of the IPv4 subnet maintenance protocols. New features introduced with NDP included address autoconfiguration, unreachability detection, and more.

Neighbor Discovery Protocol consists of five different ICMPv6 message types that include router solicitation, router advertisement, neighbor solicitation, neighbor advertisement, and redirect. Upon introduction into a subnet, a device with an IPv6 address sends a Neighbor Solicitation (NS) message to its own address. The device then listens for a response to determine if the address that it wishes to use is already in use. If another device responds to the NS message, the sender caches the address as a neighbor and attempts to generate a new address in order to avoid duplicate addresses. Additionally, the device sends out a Router Solicitation (RS)

message to locate a router that is on the local subnet. A router that is configured to forward traffic outside of the subnet then responds with a Router Advertisement (RA) message that includes a number of subnet parameters. Once this startup process is complete, the device will listen for Neighbor Advertisement (NA) messages to learn what neighbors are alive in the subnet. The device will also periodically generate NS messages addressed to cached neighbors in order to validate reachability.

An address collision resolution mechanism called Duplicate Address Detection (DAD) was formalized as part of IPv6 in RFC 2462 [77] and updated in RFC 4862 [78]. DAD is performed on all unicast IPv6 address assignments before they are assigned to an interface. This procedure uses NS and NA messages to ensure that an IPv6 address is not in use before it is assigned to an interface. To check for an address collision, the device attempting to bind the new address sends an NS message to the desired address. If the node fails to hear anything back from another node on the network, it assumes that the address is not being used and will continue assigning the address to the interface. If it hears an NA reply, it knows that the address is being used by another node on the network and must not assign that address to the interface. Additionally, a system management error must be logged. It is then up to the address generation scheme in use to determine how a new address is determined. When using SLAAC, the interface is simply disabled. According to the RFC, DAD is required no matter the means of generating an address, which includes SLAAC, Dynamic Host Configuration Protocol version 6 (DHCPv6), or manual generation.

3.7 Network Simulator 3

Network Simulator version 3 (ns-3) is an open source simulation tool published under the GNU GPLv2 license for research purposes. This particular tool performs discrete-event network simulations and can simulate point-to-point, WIFI, Carrier Sense Multiple Access (CSMA), Spectrum, and other forms of data channels. The ns-3 development team and community has also created a sizable library of default objects, and tools to create a simulated network. These objects, often referred and

named as helpers are implemented to allow users to quickly create an IPv4 or IPv6 subnet or connection with various forms of data channels.

ns-3's discrete-event simulator works by executing events scheduled in an event queue. When an event is scheduled, it is scheduled along with a time in which the event is supposed to be executed within the simulation. Once the simulation starts, the simulator begins executing the events stored in the event queue sequentially until the event queue is empty which marks the end of the simulation. When there is a time gap between the end of the current event and the start of the next event, ns-3's simulator will fast forward the simulation time so that the next event can begin immediately after the previous event has ended. This type of simulation schedules address binding and packet sending events in the desired execution order.

ns-3 provides detailed information about simulation events and output. ns-3 has an extensive logging component which permits the monitoring of event occurrence during a simulation. There are many flags that control what kinds of log information is produced when events or errors occur. In order to get more detailed output from the simulation, ns-3's tracing subsystem and logging components are used. The tracing subsystem allows the receipt information, such as when packets are received or when an item of interest changes state. The tracing subsystem's ability to capture packets is used extensively in this research's simulations.

In order for ns-3 to simulate a network, it uses nodes to represent a device connected to the simulated network. The node can be assigned attributes such as an IPv6 address and a MAC address. The nodes are contained within a node container to represent a local subnet. A single node can be contained within multiple node containers which the node can then act as a router.

3.8 Summary

Chapter 3 has presented a varying set of technologies upon which this research is built. It is due to the size of the IPv6 address space and the privacy implications of SLAAC that the original design of MT6D came to exist. By adding a combination

of key exchange and a distributed hash table to MT6D, this research is able to continue. Finally, ns-3 provides an ability to clearly understand the impacts of MT6D on network traffic, in particular NDP, without needing to execute tests at an unreasonably large scale.

Chapter 4

DHT-Based Blind Rendezvous

A key component of any distributed system is the ability to locate peers and establish a shared session. For example, in the DNS, a server's address is published within the system, and a computer requires only a URL in order to look up the address and thus find the server. DNS is a publicly accessible service with the ultimate goal of providing the widest dissemination of addresses. Some applications seek to minimize their visibility on the network in order to provide a level of secrecy or privacy. Applications that focus on security and privacy still require some means of locating peers, but must not rely on systems as widely accessible as DNS. This method of finding a peer without prior knowledge is called a *blind rendezvous* and becomes increasingly difficult as the number of possible locations grows.

An additional feature that is desirable in distributed systems is the ability for hosts to roam across the network while remaining connected to the larger system. For the purposes of this research, *roaming* is defined as the ability for some machine on the Internet to establish connections from any subnet, while still maintaining the ability to locate a resource or be located when necessary. Roaming differs from *network mobility* in that network mobility supports a movement between networks while maintaining an active connection, where roaming requires that a host remains in single subnet for the duration of a session. In DNS, roaming and mobility become challenging due to the server and cache updates required to associate a hostname

with a machine's IP address as it moves from network to network.

4.1 Motivation

A method is needed through which a machine can privately and securely share its location on the Internet without requiring direct contact with machines wishing to establish connections. As part of this solution, two machines must authenticate against each other to prevent third-party compromise of shared data. It is possible that location information could be shared between users in a secure environment before the users and their machines separate and connect to different parts of the Internet, although this sharing of static information brings with it several problems. Static distribution does not scale. If this information is to be shared with a large number of peers, the risk of compromise increases with each additional user. Since the information shared generally relates to network connectivity and includes an IP address, sharing this information statically also precludes the roaming that was discussed earlier. Another potential solution involves simply encrypting this information and sharing it on a publicly accessible single server. While this solution could work, it is vulnerable to a denial of service attack in which some adversary could prevent any of the machines from discovering connectivity information. Additionally, that single machine becomes a location that an adversary could target in an attempt to observe the connections from the machines that wish to remain hidden.

The blind rendezvous scheme presented in this chapter is a solution to these problems. This scheme permits private connection information sharing in a secure, authenticated, distributed, and dynamic manner. Strong cryptographic algorithms ensure security and authenticity, a distributed hash table avoids the risk of a single point of failure, and forcing connection information to age out ensures only current information is available as machines move around the Internet, thus enabling roaming on the network.

The proposed scheme makes use of the BitTorrent Mainline DHT as a mechanism for storing encrypted information as a form of escrow. In 2014, the average daily

number of DHT-peers was on the order of 7.5 million nodes, which were distributed among a large number of different geopolitical locations [55]. The scheme makes use of the immense size of this peer-to-peer network to provide both resilience and suitable cover traffic to ensure the security, reliability, and privacy of the participants.

4.2 DHT-Based Information Exchange

One of the largest open distributed systems in use on today's Internet is the BitTorrent DHT, which is used for establishing connections between peers in the Peer to Peer (P2P) BitTorrent system. This immense open distributed system can be leveraged to facilitate blind rendezvous and information exchange. The information exchange provides the ability to securely and dynamically exchange configuration information in order to support the scaling and mobility desired in a number of privacy preserving protocols. In order to exchange this information, one machine generates a descriptor and a message that are then published into the DHT for another machine to discover.

4.2.1 DHT Descriptor Generation

The BitTorrent Mainline DHT is an implementation of the Kademlia DHT algorithm, and as such uses an opaque 160-bit address space for publishing and retrieving information [50]. In normal operation, the DHT descriptor or "infohash" is generated by taking the SHA1 hash of the value of the info key from the `.torrent` file [27]. This scheme relies on a keyed hash of time construction where the key is generated using an Elliptic Curve Diffie-Hellman (ECDH) function. Specifically, the scheme utilizes the Curve25519 Diffie-Hellman function [17].

There are two protocol parameters which need to be agreed upon prior to deployment:

- H – A cryptographically strong hash function with a digest length of at least 160 bits. (e.g., SHA256)

- n – The length of the validity period for a descriptor.

The strength of the hash function is dependent on the computing power of the devices utilizing the scheme versus the security requirements. Similarly, the length of n will vary depending on the constraints of the devices. A longer period requires fewer computational and network resources, but creates a larger window of attack if an adversary is able to identify the node hosting the DHT message.

For two users with public-private keypairs (X, x) and (Y, y), the shared moving-target DHT descriptor d at any time t within the time window T of length n is generated as:

$$s = X \cdot y = Y \cdot x \tag{4.1a}$$

$$T = t - (t \mod n) \tag{4.1b}$$

$$d_T = H(s||T)_{0 \to 159} \tag{4.1c}$$

where $||$ denotes concatenation and $\cdot$ denotes scalar multiplication over the elliptic curve Curve25519. This descriptor can then be used to publish information which is boxed using the scheme described in the following section. Due to the features provided by Curve25519, both users are able to generate the same descriptor, and can therefore publish and retrieve information though the DHT under that value.

4.2.2 Port Stuffing Information

The BitTorrent Mainline DHT was designed to facilitate trackerless BitTorrent operation. Peers utilize the infohash of the torrent they wish to download to locate other peers downloading the same torrent, and publish their IP address and port into the DHT to allow other peers to locate them. This leads to a very spartan design where the only data stored under the infohash descriptor is a list of IP addresses and port numbers. The handshake mechanism of retrieving a `token` value based on the IP address and a changing secret value makes it impossible or impractical to store data under spoofed IP addresses. Instead, there exists a simple method of stuffing

data into the port section of the contact information that allows for exchange of a limited quantity of arbitrary information.

In order to send a B byte message, generate B unique random numbers over the interval $[0, 255]$ as an ordered list $\{p_0, p_1, \ldots p_B\}$. For each byte of the message b_i, generate a 16-bit port as $(p_i << 8|b_i)$. Send an `announce_peer` message to the DHT under infohash d for the current time interval with that port. This allows the user to send up to 256-byte arbitrary messages to be stored in the DHT for some interval of time.

Users can leverage this technique to send arbitrary data of up to 256 bytes per IP address into the DHT. A user should ensure adequate security of the data in case an eavesdropper discovers the DHT descriptor by snooping on Internet traffic between unsecured nodes. In the proof of concept implementation, the *libsodium* [30] library provides all cryptographic functions. *libsodium* implements Bernstein's XSalsa20 stream cipher [18] and the Poly1305 authenticator [16]. The software relies on keys generated from the Curve25519 function for descriptor generation and a nonce based on the generated descriptor d. These algorithms fulfill all of the requirements for XSalsa20-Poly1305 nonces [16]. This results in an overhead of 16 bytes for the authenticator tag, leaving 240 bytes for the user's application.

4.2.3 Arbitrary Data Storage Design

While the limitation of 256 bytes is not problematic for many applications which only need to exchange a limited amount of information to establish a session or to send a pre-agreed message, it might be limiting for other applications which need to send more data in a time interval. Fortunately, proposals have been put forward to extend the BitTorrent DHT to allow for arbitrary data storage of immutable and mutable objects [64]. If the proposed extensions are widely adopted, this scheme can be modified to allow much more efficient storage of secure data in the DHT.

The proposal would allow for storing up to 1000 bytes of encoded data in the DHT signed by an Ed25519 key and stored under the SHA1 hash of that key. In order to adapt the scheme presented here to this newly proposed scheme, the descriptor

generation scheme is modified to generate a shared Ed25519 private key for each time interval, and messages are published under the SHA1 hash of that key's public key. This modification is achieved by requiring that H, the hash function, has an output length of at least 256 bits (such as SHA256). The shared secret s would be generated as before, but instead of generating a descriptor directly, the scheme generates a shared private key: $sPK_T = H(s||T)_{0\rightarrow 255}$ which is then converted to an Ed25519 private-public key pair using the method described by Bernstein [19]. The users of this scheme can then publish and retrieve data under that keypair's hash as described in BitTorrent Extension Proposal (BEP) 44 [64].

4.3 Analysis

Unlike the IPv6 Internet which uses a hierarchical addressing scheme where portions of the address are reserved for routing information, the DHT keyspace is flat. This changes the size of the keyspace from the size of the subnet (2^{64}) to the size of the entire address (2^{160}). This both improves the security margin by requiring the attacker to search a much larger address space and side-steps a difficult problem in the deployment of MT6D: locating the routing information of the other party.

H, the hashing function used to generate the descriptor based on a shared secret key and time, must be cryptographically secure in order to draw some conclusions about the DHT descriptors that it will generate. The Kademlia DHT used by BitTorrent defines the closeness of two nodes and the closeness of a node to an infohash by an XOR metric [52]. The distance is calculated as: $\text{distance}(A, B) = |A \oplus B|$. The strong hash function should not permit any relation between outputs; on average each bit has a 50% probability of flipping between inputs, therefore it is expected that the average distance metric between any two descriptors should be $\frac{160}{2} = 80$. This ensures that consecutive descriptors are unlikely to be placed in the same DHT bucket, and are likely to be stored on very different nodes within the network. A small example demonstrating this property is shown in Table 4.1 where it is easy to see that there is an average distance of 83.25, which is quite close to the ideal value

Table 4.1: Sample configuration where user A calculates five sequential descriptors using user B's public key. The XOR metric is calculated for each descriptor relative to the previous descriptor.

User	Key Type	Key Value
A	Secret	`866e8d6a73cbf518cf79be36d331e7c1d4a7e0de82a46f1a68fe1335d3c6419c`
B	Public	`ef7f40dd3a1dde583f0a4e5b9e0d3716801812271cc162a5d4f5910f6be7e456`

T	Descriptor	Metric to Previous
0	`0feb53b27f98590d0bb62a04c4e13ba65a52a780`	–
1	`98590d0bb62a04c4e13ba65a52a780e389e9c84a`	90
2	`2a04c4e13ba65a52a780e389e9c84a10a255b06a`	87
3	`a65a52a780e389e9c84a10a255b06a8afe24982a`	80
4	`e389e9c84a10a255b06a8afe24982afebe33ea76`	76

of 80. Messages from any two consecutive time intervals should be stored on different machines located using almost completely different lookup paths. The immense size and churn of the network [80] provides an expectation that the protocol will make use of very different portions of the network. This expectation grows even stronger with the adoption of DHT Security Extensions [63], which prevents an attacker from executing a Sybil attack and becoming the DHT node where a message will be published or has been published. In a Sybil attack, an attacker generates a large number of pseudonymous identities within the network, thus gaining a disproportionately large influence.

4.4 Testing the Configuration Exchange

To demonstrate the viability of the scheme, a proof of concept implementation was built in C on a 64-bit Debian 7 "wheezy" distribution of Linux.

4.4.1 Proof of Concept Infrastructure

The heart of the scheme lies in its use of cryptography and a common storage location on the Internet in the form of a DHT. For the proof of concept implementa-

tion, the Sodium crypto library [30] provides all cryptographic calculation functions. Sodium is a software library that provides a number of cryptographic functions, including key generation, encryption, decryption, and message authentication.

In a full scale implementation, the scheme relies on the size and availability provided by the BitTorrent Mainline DHT, but for proof of concept implementation, a local key-value storage server running redis [68] is used. Redis is a key-value storage software that provides a very easy to use API in which key-value pairs can very easily be stored and retrieved. Redis provides a capability similar to that of the DHT while maintaining control of research data and providing the ability to analyze data while it is present in the DHT analog.

4.4.2 Key Pair Generation

As discussed in Section 4.2, the root of the cryptographic functions that are used in this scheme is a key pair. In a production implementation, the implementer would be responsible for devising a solution for the generation and distribution of these keys. In this specific implementation, the Sodium cryptographic library's `crypto_box_keypair()` function generates keys, which are distributed by creating a simple text-based keystore for each machine in the simulated network. This key distribution is even further simplified in this implementation due to the fact that the clients and server consist of multiple processes running on the same machine.

4.4.3 Server Implementation

For proof of concept, the client and server do not have any functionality beyond the ability for the server to store some arbitrary piece of data in the hash table that is then retrieved by the client. Beyond this simple exchange of data, the server does not provide what could be considered a traditional service to the client. In a production environment though, the small piece of data would be the key to either locating a hidden service or a piece of information required to establish some further connection between a client and server. In this larger work, this small piece of data

is the key information required to begin the establishment of an MT6D connection.

As discussed earlier, this implementation requires that the server generates a different message for each client, even if each client should receive the same unencrypted message. This is due to the fact that the server uses the client's public key in order to encrypt the original message. Within the Sodium library, a function named `crypto_box_easy()` is provided that allows easy encryption of the message for publication to the DHT. Additionally, the Sodium provided function `crypto_hash_sha256()` generates the DHT key or descriptor, introduced earlier as d. A shared secret is calculated using the `scalar_mult()` function over the private key of the server and the public key of the client. This shared secret value is then concatenated with the current time window, T, which is then hashed to determine the DHT key for the current time window, d_T. The server publishes the key (d_T) and the encrypted message into the distributed hash table or in this case the key value server.

4.4.4 Client Implementation

In addition to the server, a proof of concept client was implemented that requests the data from the distributed hash table that the server put there for it. To do this, the client follows the same steps that the server took in reverse. First, the client generates d_T by concatenating the shared secret between the client and the server (generated using the `scalar_mult(q, n, p)` function over the client's private key and the server's public key) and the time of the current time-interval and hashing it using the `crypto_hash_sha256()` function. The client queries the DHT with d_T to discover if there is a current time interval message from the server waiting. If such a message exists, the DHT returns the message to the client. If a message doesn't exist and the keys have been used correctly, it is likely that the message has expired and been aged out of the DHT. The client uses `crypto_box_open_easy()` to retrieve the contents of the message. At this point, the client can use the contents of the message to execute some further task, whether that be beginning the establishment of a moving target defense connection or establishing a connection to some hidden

service.

4.5 Summary

This chapter has introduced a technique that leverages strong cryptographic functions and a large scale DHT in order to exchange some key piece of information. This process could be considered somewhat similar to a dead drop, where someone hides something important at a pre-determined location for some other person to come retrieve at a later point in time. In the case of this scheme, the pre-determined location is calculated on the fly and changes as time progresses through time-intervals. Since this scheme is based on public/private key pairs, peers can also be sure that the party with whom they are communicating is correct. This scheme is the foundation upon which the next chapter's MT6D key exchange process was built.

Chapter 5

Configuration Exchange in MT6D

Exchanging keying information has been one of the main weaknesses of network-based moving target defense systems since their inception. In the past, establishing a moving target connection required that two peers exchange some shared key and their predicted subnet manually in order to establish a connection. This requirement caused a number of problems, including a limitation on network mobility, the inherent risk in exchanging static security information, and the impracticality of attempting to scale this method to a large number of hosts. In today's Internet, it is normal for a machine to move between subnets on a fairly regular basis. Consider your smartphone or your laptop and the number of subnets that it connects to in any given day. Due to the hierarchical nature of Internet routing, your moving target defense peer would need to know each subnet that you would be connected to throughout the day in order to establish a connection to you. Additionally, statically defined and manually distributed configurations preclude the ability to modify configurations on the fly based on potential compromises that may be detected. In an ideal scenario, the moving target defense configuration could be modified mid-session due to the movement of an endpoint through the network or in order to work around a compromise. Finally, the manual distribution of configurations does not scale. As the number of nodes with secret information increases, so to does the risk of compromise increase as well. Finally, the configuration of every node in a large scale moving

target network presents an immense logistical challenge. This challenge includes not only the configuration of the nodes, but also the out-of-band distribution of configuration information. If it is desired that a moving target defense scenario involve a large number of peers, some automated means of distribution must be established.

This chapter brings together the address changing techniques of MT6D that were presented in Section 3.3 and the blind rendezvous scheme that was presented in Chapter 4. It is through the combination of these technologies that a network-based moving target defense system gains the capability to dynamically and securely exchange configuration information. Exchanging configuration information through a blind rendezvous technique enables a roaming capability, an ability to scale, and a dynamism that is not available in any current moving target defense system and reduces the amount of pre-shared information to only a public key. This new scheme relies on the size, resilience, and anonymous nature of the BitTorrent Distributed Hash Table to improve security, privacy, and anonymity.

5.1 Design

Establishing an MT6D session requires that each end-point has the necessary configuration information. As introduced in Section 3.3, MT6D addresses are calculated by hashing a host's EUI-64 IID, a shared session key, and a timestamp [33]. The values are concatenated and hashed using the form:

$$IID'_{x(t_i)} = H[IID_x||K_S||t_i]_{0\rightarrow 63} \tag{5.1}$$

where $IID'_{x(t_i)}$ represents the MT6D IID for host x at time t_i, IID_x represents the EUI-64 or statically defined IID from host x, K_S represents a shared session key, and t_i represents the time at instance i. $||$ denotes concatenation, and H is some cryptographically strong hashing algorithm with a result longer than 64 bits, such as SHA256. The first 64 bits of the hash are then concatenated with the 64-bit network address of host x, resulting in a complete 128-bit MT6D generated address.

While the original implementation of MT6D required a manual exchange of con-

figuration information and the generation of a static configuration file to hold said information, this work adds the ability to dynamically generate this configuration information and share it over the network in a private but authenticated manner. The move towards dynamically generated and automatically distributed configurations helps to overcome limiting factors inherent in the initial design of MT6D. The modifications to MT6D include a move from a statically defined symmetric key to a randomly generated key that changes at some pre-determined interval. Changing the symmetric key ensures that a captured key is only valid for a short period of time, thus limiting exposure in the case of a compromise.

A number of additional minor modifications to the MT6D protocol were required in order to overcome several other issues. These modifications are slightly different depending on the role of the endpoint in the network conversation and will be described below. For purposes of this work, the server is defined as the machine that is generating the connection information and the client as the machine that is consuming that configuration information and establishing a connection to the server. There is no requirement that there are multiple clients connecting to a server, just that the server is the machine generating the configuration information.

5.1.1 Server

It is the server's responsibility to ensure that configuration data is updated and published to the DHT on a periodic basis. This time period should be based on the specific situation in which MT6D is being implemented. In an environment where there is greater risk to compromise or attack, the scheme should be tuned to rotate configuration information more often. This time period, T, is defined as the beginning of the time window during which a specific configuration will be valid. T is of some pre-configured length n. In order to ensure configuration synchronization between hosts, T is calculated using $(now - (now \bmod n))$. This use of modulo arithmetic in combination with time forces time windows to begin on regular intervals, thus ensuring T is the same between hosts. Using T as a component within the DHT descriptor also forces the configuration messages to expire periodically.

Publishing information according to the DHT-based blind rendezvous scheme described in Chapter 4 requires that the server generates a descriptor for time period T, d_T, and a message m. d_T is calculated by the server using Equation 5.2a, where H is a strong hashing algorithm such as SHA256, S_{pri} is the server's private key, C^i_{pub} is the public key for client C^i, and T defines the time period. The server then generates the message, m, by using Equation 5.2b where $SeedIPv6_S$ is an IPv6 address which is the concatenation of the server's IPv6 subnet and a randomly generated IID. K is the MT6D symmetric key, Rot is the address rotation period, and $Nonce$ and MAC are generated by the encryption algorithms to prevent replay attacks. Notice also that in the generation of the message, the scheme encrypts the configuration data with both the server's private key as a digital signature and the client's public key to ensure that only the client can decrypt the message. The server then publishes message m using descriptor d_T into the DHT for future retrieval by the client.

$$d_T = H(S_{pri} \cdot C^i_{pub})||T)_{0 \rightarrow 159} \tag{5.2a}$$

$$m = Nonce||MAC||E_{C^i_{pub}}(E_{S_{pri}}(E_{Nonce}(SeedIPv6_S||K||Rot))) \tag{5.2b}$$

The original design of MT6D relied on the idea that both endpoints have knowledge of their peer's EUI-64 calculated IID, subnet, pre-shared key, and MT6D rotation time interval. The goal was to improve the protocol by requiring that as little data as possible be known by each endpoint prior to connection. By applying the DHT blind rendezvous scheme to MT6D, this work requires that no specific network data be known by either end-point prior to connection. The blind rendezvous scheme reduced the shared information requirement so that only public keys need be shared between end-points.

Since the server no longer needs prior knowledge of the client's EUI-64 IID or subnet, and in fact could not know the specific subnet from which the client would connect, the scheme cannot use the same methods applied in the original design of MT6D. Instead, it must use some values for client subnet and IID that can be known to all parties. This problem is solved by using a technique similar to that described in

Section 4.2 for the generation of the descriptor. The client's seed IID is generated as shown in Equation 5.3 where (S_{pub}, S_{pri}) is the public/private key pair of the server, (C_{pub}, C_{pri}) is the public/private key pair of the client, and K is a key that has been pseudo-randomly generated by the server and was retrieved by the client from the DHT. Using this method to generate the client's seed IID, C_{IID}, ensures that both endpoints of the conversation use the same seed information when calculating MT6D addresses, but ensures that each of the clients use a different seed and thus generate different IPv6 addresses even if all other address generation elements are common.

$$C_{IID} = H((S_{pri} \cdot C_{pub})||K)_{0 \to 63} = H((S_{pub} \cdot C_{pri})||K)_{0 \to 63} \tag{5.3}$$

MT6D's use of statically configured keys brings with it a number of issues, including insecurity and the inability to scale. Should it happen that some adversary gains access to the key, there is no way to generate and distribute a new key, so communications must cease until a new key could be exchanged out-of-band. A compromised key provides some unauthorized party the ability to generate the correct MT6D addresses, thus allowing them to listen on those addresses, or worse impersonate one of the endpoints. Additionally, it is human nature to select a weak key that is potentially guessable by some adversary, increasing the risk of potential dictionary or rainbow table attacks against the scheme. Through modifications to MT6D proposed here, MT6D transitions from using statically defined, user selected keys to using randomly generated keys that expire at fixed intervals and are shared via the BitTorrent DHT. With an established client IID, rotation period, and key, the server now has the ability to calculate MT6D IIDs for the client.

5.1.2 Client

The client obtains configuration information from the DHT by querying for the descriptor that is calculated with $H(S_{pub} \cdot C^i_{pri})||T)_{0 \to 159}$, which is the inverse of Equation 5.2a, but because of the properties of ECDH results in the same value. Upon receipt of the shared message, m, the client decrypts and validates the message using its private key, the server's public key, and the nonce that was provided in the

message. As long as there is a message somewhere in the DHT that is defined by that descriptor, the client will retrieve it. In the case that the client requests a descriptor on the boundary between two different T windows, it may fail to find the current descriptor or will retrieve expired data. Once it decrypts the data and discovers that it is expired, it simply searches for a descriptor with the new T value. T should be measured in minutes or hours, so the cost of less than a second to query, retrieve, and decrypt an expired message then re-query, retrieve, and decrypt the current time period's message is minimally impactful. Once decrypted, the retrieved message contains all of the configuration information required to calculate its own MT6D addresses, namely the key (K), the rotation period (Rot), and the server's seed IID ($SeedIPv6_S$). Additionally, the client calculates its own MT6D seed IID by using Equation 5.3 with the server's public key and its own private key.

In its current implementation, MT6D relies on Linux tunnel interfaces in order to route traffic through the MT6D process. The client must know the tunnel address that the server is using in order to ensure that traffic is routed through MT6D towards the server. Since this address must be known to the client, it is require that this address be included in the configuration information that is sent through the DHT. A randomly generated address has the benefit of adding additional entropy into the system and ensuring that multiple servers residing on the same subnet do not interfere with each other. It is for this reason that the server generates a random value for its $SeedIPv6_S$ address rather than using a static value.

The intent of this scheme is that this MT6D configuration will only be used as a means of conducting an introduction between the two hosts. On the server, this set of addresses will be common among all clients that are known to it. This limitation is introduced to save resources on the server, requiring that it only maintain a single set of MT6D addresses for initial contact with clients. During the introduction, the hosts will authenticate each other using their public and private keys and agree on a new session key. This new session key will then result in a new set of MT6D addresses that will only be valid between the specific client and server. The server will then maintain a separate set of MT6D addresses for each of the clients that have established sessions with it and an additional set of addresses for introductions. This

idea will be discussed in much greater detail in Chapter 6.

5.2 Implementation Results

The proof of concept implementation of the MT6D key exchange scheme leverages the power of a number of C libraries. Much of the implementation is similar to the proof of concept implementation described in Section 4.4. Elements of the libraries, *libsodium* [30], *libev* [49], and *redis* [68] provide the majority of the external code requirements. Sodium provides all of the cryptographic functions, including key pair generation, encryption, signing, and decryption. Libev is an event handler that manages the run loops, including address rotation, configuration rotation, and manages an exit handler so that the program can exit gracefully. Redis is a key/value server that is used as an analog to the BitTorrent DHT. Rather than potentially causing issues on the DHT, data is kept on an internal research network for proof of concept and testing.

Figure 5.1 provides a snippet of output from the server. The first three lines of the figure provide the three keys that the server has access to, namely its secret key, its public key, and its peer's public key. After publication to the DHT, the server shows the descriptor that was used as an index into the DHT and the message that was published for a client to retrieve. The descriptor and message in the figure are generated using Equation 5.2a and Equation 5.2b.

Figures 5.2 and 5.3 show a sample execution of both the server and the client,

```
My Secret: 94197ec9bbaa690954a8e2d18c7b93aa032229b56289cae2a0ac8ba1f4638bc3
My Public: 7a1294c77b0c33db228d0572b7ea32117cfce4bd854594a3000faa5c0a537206
Peer Public: 175abafc765430c67893adbbd1dd3a2a4edabd81a03b208bd97135c52fcf0436
Publish to DHT.
Descriptor: 89b5b7fe806fa26c9eb22d933f53ca46671ca8cb
Server Message: da4e54c638ce622e8539fcb1a195990b267547521bf649698d9c9d97a623175
12abc91287f6cc2edcca0ccad89e594dc0598e2b58a869cea5017178b4904551c1f
```

Figure 5.1: Server output shows key elements of the key exchange scheme.

demonstrating the effect that the configuration rotation has on the client's ability to find the server. For brevity, only the addresses that are assigned to the server are shown, while in reality, both the client and server are calculating addresses for the client as well. In Figure 5.2, time T is shown by the addresses in the top portion of the figure, while time $T + 1$ is shown below the configuration rotation notification. $T + 1$ represents the next time interval for purposes of configuration data aging.

```
Addr: 2001:468:c80:c111:e077:8a52:e427:450c
Addr: 2001:468:c80:c111:a267:21de:97cb:4c3b
Addr: 2001:468:c80:c111:6e9f:44b6:e635:f5c4
Addr: 2001:468:c80:c111:4dd8:79f7:d110:fb3c
Addr: 2001:468:c80:c111:e47c:2aa7:3d07:adf3
Addr: 2001:468:c80:c111:e224:31e3:646e:b85
Addr: 2001:468:c80:c111:9e8a:c35d:867:4ee1
Addr: 2001:468:c80:c111:57be:f412:51a5:7280
Addr: 2001:468:c80:c111:718f:356c:28b8:e653
Addr: 2001:468:c80:c111:b0aa:6706:e6b3:a97
Addr: 2001:468:c80:c111:9f13:9d68:7030:559c
Rotating configuration information (new key).
Addr: 2001:468:c80:c111:864f:f74:b675:971b
Addr: 2001:468:c80:c111:f2f8:5faf:61e1:5e95
Addr: 2001:468:c80:c111:fc0b:e798:d34e:8885
```

Figure 5.2: The server's MT6D addresses during time T and $T + 1$.

The data in Figure 5.2 becomes more meaningful when compared with the addresses that are displayed in Figure 5.3. The client begins execution during time period T, quickly pulls configuration information from the DHT, and immediately synchronizes MT6D addresses with the server. The client's first calculated address matches the third address that the server has calculated. Addresses are continuously in sync until the server rotates configuration. Upon rotation, the server begins to use a new session key which the client does not have. After the address that ends with 559c, the client can no longer calculate the correct address and hence cannot communicate with the server. If the client simply requests the configuration for time period $T+1$ from the DHT, it will acquire the new session key and regain the ability to synchronize with the server.

```
Starting Client
Addr: 2001:468:c80:c111:6e9f:44b6:e635:f5c4
Addr: 2001:468:c80:c111:4dd8:79f7:d110:fb3c
Addr: 2001:468:c80:c111:e47c:2aa7:3d07:adf3
Addr: 2001:468:c80:c111:e224:31e3:646e:b85
Addr: 2001:468:c80:c111:9e8a:c35d:867:4ee1
Addr: 2001:468:c80:c111:57be:f412:51a5:7280
Addr: 2001:468:c80:c111:718f:356c:28b8:e653
Addr: 2001:468:c80:c111:b0aa:6706:e6b3:a97
Addr: 2001:468:c80:c111:9f13:9d68:7030:559c
Addr: 2001:468:c80:c111:9713:e6cb:9846:8ad0
Addr: 2001:468:c80:c111:34d3:a5f5:8ab4:2979
Addr: 2001:468:c80:c111:278a:43c1:2ae8:3a6e
Addr: 2001:468:c80:c111:cae9:9cae:ce0e:75c7
```

Figure 5.3: The client's MT6D addresses during time T and $T + 1$.

Applying the blind rendezvous methodology from Chapter 4 to MT6D has enabled a roaming capability for clients as well as alleviated the problem of key exchange and key distribution. This helps to allow the MT6D network to scale from a purely peer-to-peer tunnel architecture to one where there are many clients communicating with each other and with servers which provide services.

5.3 Summary

This chapter presented modifications to the MT6D protocol which leverage the DHT-based blind rendezvous scheme that was presented in Chapter 4. By applying this scheme, MT6D now has the beginnings of a network-based moving target defense system that enables roaming, is scalable, and is increasingly secure. By reducing the amount of data shared between hosts to simply a public key, the research simplifies the distribution of moving target defense configuration information and reduce the risk of compromising said data.

The work additionally demonstrated an implementation of the ideas through the modification of MT6D, showing that the ideas are in fact feasible. The proof of

concept successfully allows a highly scalable yet secure exchange of moving target defense configuration information between peers. While the proof of concept is limited in functionality, it clearly demonstrates the first steps towards establishing a fully functional MT6D server. In the next chapter, this foundational work will be extended into an MT6D server.

Chapter 6

Implementing and Testing the MT6D Server

Originally, MT6D was developed for peer to peer use and relied heavily on static configuration files for all of the address generation data. This chapter builds upon foundational work presented in Chapters 4 and 5 to extend MT6D in support of a client/server network [56, 57]. Doing so requires the exchange of dynamic configuration information, the ability to scale to a large number of clients, and some means to maintain a large number of client connections concurrently. This chapter completes the integration of these ideas and presents a fully functional network-based moving target defense server solution.

6.1 Server Design

Previous implementations of MT6D involved a heavy reliance on manually defined static configuration information. In the development of the server, it became necessary to permit dynamic configuration of the MT6D endpoints, due primarily to the fact that attempting to manually configure a large number of clients would be impractical and would not provide any means of supporting network mobility. By applying the DHT blind rendezvous techniques presented in Chapter 4 and apply-

ing them to MT6D as described in Chapter 5, MT6D took the first steps towards enabling the scaling that is required by a moving target defense server. Beyond this exchange of initial configuration information through the DHT, this section presents the architecture, messages, and procedures required to establish dynamically created MT6D connections between a server and some number of clients. This section begins with a high level overview of the MT6D server and follows with a much more detailed description, including server startup and session establishment protocol.

6.1.1 High Level Overview

In an effort to maintain a level of separation between the clients that may connect to a common server, separate MT6D connections are established between each client and the server. This is as an alternative to having the server maintain a single MT6D instance that is connected to by all possible clients. Building the system so that clients all connect to a single MT6D instance simplifies many of the session establishment requirements, but reduces the level of anonymity and privacy that each client enjoys. Maintaining separate MT6D instances for each client, helps to ensure that the clients are not able to discover each other on the network by simply listening to network traffic. In order to successfully maintain separate connections, the server is required to conduct separate MT6D address calculations for each client connection and bind those addresses to virtual interfaces that are instantiated upon client request. The establishment of these separate connections requires some common means of finding the server on the network, which is accomplished by maintaining what is referred to as an introduction interface on the server and publishing the requisite information into the DHT according to the scheme presented in Chapter 5. The introduction interface is maintained by the server's main MT6D process and is monitored for connection requests coming from clients. Upon receiving a connection request, the server spins up a new MT6D virtual interface that will only exist for the duration of the session with the given client and is only used for communication with that client. The system presented here relies heavily on public key infrastructure not only in the ways that have been described already in Chapters 4 and 5, but also

as a means to encrypt and authenticate all of the scheme's messages. Any session establishment message, including those published to the DHT, are both signed by the originator's private key and encrypted using the intended recipient's public key. The combination of these keys provides an additional layer of protection from man in the middle and replay attacks.

This implementation of an MT6D server leverages the power of Linux tunnel interfaces to ease the scaling of MT6D connections. In Linux, a tunnel interface is a virtual network interface that is attached to some physical interface, but can be treated as a normal network interface by any running processes. This work uses tunnel interfaces behind the physical Ethernet interface in order to maintain a single physical connection to the network, while still providing the ability to increase the number of MT6D generated addresses that are being monitored and to keep those addresses logically separated by session. Because tunnel interfaces are virtual, they can be spawned and destroyed through the flow of the program as the number of concurrently communicating clients increases or decreases. In the implementation of the server, it maintains a single tunnel interface for configuration requests and responses, and adds a new tunnel interface for communication with each client for the duration of its session.

The establishment of a session requires that the server first achieves a steady-state in which it is available for connections. Also required is a protocol through which the clients can communicate with the server in order to establish these sessions. The next sections will discuss server setup and the session establishment protocol in much greater detail.

6.1.2 Server Startup

When the server starts, the steps shown in Algorithm 1 are completed to get to a steady-state where the server is prepared to accept connections from clients. Because there are multiple event triggers that must be monitored, the server uses an event loop library for the maintenance of event timers, traffic watchers, and signal watchers. Since the server could end up binding a large number of IPv6 addresses to

Algorithm 1 Server Start-up Algorithm

1: Start SIGINT watcher
2: Create introduction interface (mt6d0)
3: Generate and bind tunnel interface IP
4: Start address rotation timer
5: Start configuration publication timer
6: Start outbound tunnel interface watcher
7: Start inbound UDP traffic watcher

the physical Ethernet interface, it registers a SIGINT watcher that ensures that it can exit gracefully and remove all of the bound addresses. With the signal watcher in place, the server creates the introduction tunnel interface, which is used for all session establishment communication. The server assigns this interface a self-generated IPv6 address that is a concatenation of the 64-bit IPv6 subnet address, which was retrieved from the Ethernet interface, and 64 randomly generated bits. In Section 5.1, the server's self-assigned address is referred to as $SeedIPv6_S$. This address will be the destination address of all configuration request messages coming from any of the clients as they traverse the MT6D established tunnels. The server also generates a 128-bit introduction session key, which is used in combination with the tunnel interface IID for MT6D address generation calculations as discussed in Section 3.3. With these pieces in place, the server starts two timers and two watchers. The first timer is used for address changes according to the MT6D algorithm and defaults to three seconds. At each timer expiration, the server generates a new MT6D address for the time period and binds it to the Ethernet interface. The second timer is used for the rotation of introduction session configuration information which was discussed in detail in Chapter 4 and will be discussed further in Section 6.1.3. The server also generates two input/output watchers that look for MT6D data inbound from or outbound to the network. The outbound tunnel interface watcher ensures that traffic that is destined for an MT6D host is encapsulated in a UDP header with the current MT6D generated ports. The datagram is then encapsulated into an IPv6 header with the current time period's MT6D generated source and destination IPv6 addresses. The server uses the inbound UDP traffic watcher to catch any UDP

traffic that is inbound from an MT6D host and arriving addressed to an MT6D IPv6 address and current MT6D UDP port. This traffic is then routed through the MT6D process in order to decapsulate the IPv6 packet which is then passed to the traditional network stack for processing.

Once the server has completed its startup procedures and is in its steady-state, it forces an expiration of the configuration publication timer, which triggers the steps presented in the next section and Algorithm 2.

6.1.3 Configuration Updates

As described in the previous section, the server generates an introduction session key during start-up that is applied to the MT6D address calculations in the main MT6D process. Periodically, the server re-generates this session key in order to ensure that only current clients are able to discover configuration information within the DHT. This period is described as the introduction session configuration period, and is monitored by the configuration publication timer. At each timer trigger, which is fifteen minutes by default, the server executes the steps described in Algorithm 2, shown below. Simply put, the server generates a new introduction session key and publishes a new message into the DHT for each of the registered clients to retrieve according to the process discussed in Chapter 5. Separate messages are used for each of the potential clients so that the strength of public key encryption may be leveraged to ensure that only authorized clients are able to retrieve and decrypt messages. Since each of these messages is both signed and encrypted, clients can be certain that messages retrieved from the DHT are for them and are definitely from the server. A server must have prior knowledge of all potential clients that may want to connect to it, which is referred to as *registering with the server*. This registration is simply the act of providing the server some identifying information that allows it to procure the client's public key. In the proof of concept implementation, the server maintains a copy of each client's public key in a text-based key store.

Algorithm 2 Configuration Publication Algorithm

for each configuration time period (T) **do**
    Generate introduction key (K)
    Build/Reconfigure introduction interface
    for each registered client (C^i) **do**
        Calculate the descriptor:
        $d_T = H(S_{pri} \cdot C^i_{pub})||T)_{0 \rightarrow 159}$
        Randomly generate a message nonce (N)
        Calculate the message:
        $m = N||MAC||E_{C^i_{pub}}(E_{S_{pri}}(E_N(Pfx_S||K||Rot)))$
        Publish m at d_T in DHT
    end for
end for

6.1.4 Session Establishment

In order to fully support the establishment of dynamic MT6D sessions, development of a session establishment protocol is required. The protocol is a layer 4 messaging protocol that uses an IPv6 next header value of 0x254, which defines experimental protocols. By using a next header value of 0x254, the machines are able to easily differentiate traffic on the endpoints as it is received. Currently, only three possible message types are defined as part of the session establishment protocol. Message ID 0 is a configuration request message, message ID 1 is a configuration response message, and message ID 2 is a session teardown message. Each message begins with the 8-bit value that is its identification, and is followed by any other data pertinent to that message. In the following sections, the content of each of the protocol's message types and their use in a standard message flow are described in much greater detail.

Session Establishment Messages

The configuration request message is identified as message ID 0, and is sent by the client to request a session configuration from the server, and thus is the message that begins the entire process of session establishment. Not only does it begin the

session establishment process, but it gives the client an opportunity to tell the server where it is located on the Internet, which client it is, and what seed IID it will use to calculate MT6D addresses. As can be seen in Figure 6.1, the configuration request message is a 36-byte message that consists of an 8-bit message ID, a 24-bit client ID, and two 128-bit IPv6 addresses. The first address identifies the seed address that the client will use in the calculation of MT6D addresses and the second defines the tunnel address that the client will use in order to route traffic into the MT6D process.

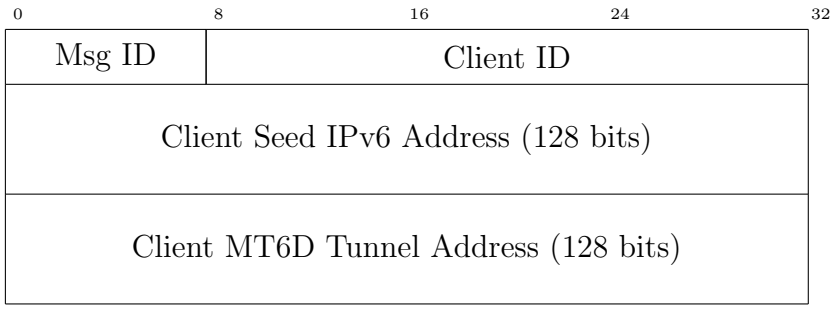

Figure 6.1: The Configuration Request Message (MSG ID 0)

The configuration response message, identified as message ID 1, is sent by the server in response to a configuration request message. The response message, which is depicted in Figure 6.2, is a 36-byte message that consists of an 8-bit message ID, a 24-bit address rotation period, a 128-bit tunnel IP address, and a 128-bit session key. As with the client's configuration request message, the server maintains a tunnel address which is given to the client to ensure MT6D traffic is routed through the MT6D tunnel.

The final message that is currently defined is identified as message ID 2, and is best described as a session teardown message. The message is 4-bytes long and is depicted in Figure 6.3. The session teardown message consists of only the message ID and client ID. The message is sent by either the client or the server when they are ready to tear down the current session. While this message is not necessary because of timers that are integrated into both the client and the server, sending it helps to close sessions more quickly. Use of this message not only helps to save

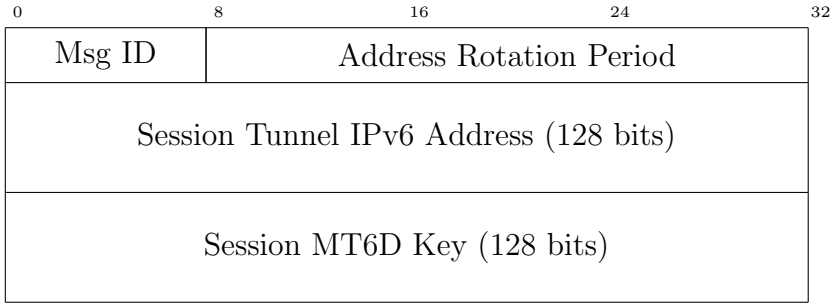

Figure 6.2: The Configuration Response Message (MSG ID 1)

network resources, but also saves computation resources on the communicating peers by halting MT6D address calculations sooner.

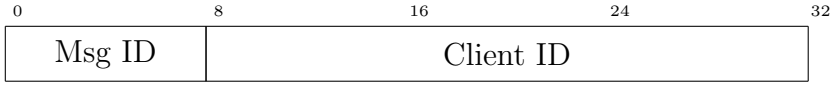

Figure 6.3: The Session Teardown Message (MSG ID 2)

Session Flow

In an effort to more clearly describe the processes required for a client to establish a connection with an MT6D server, a message sequence chart is presented as Figure 6.4. Prior to the first message in the chart being sent by the client, the server has completed its startup procedures and is now in its steady-state, listening on its introduction interface for configuration request messages from its registered clients.

When the client is prepared to request a session from the server, it begins by querying the DHT for introduction configuration information. As described in Chapter 5, the client generates a descriptor using the inverse of Equation 5.2a. Based on the properties of Elliptic Curve Diffie-Hellman [20], the scheme generates the same shared secret that the server was able to generate by using the client's private key and the server's public key as shown in Equation 6.1. This shared secret is the descriptor that describes the location with the DHT at which the server published a message containing introduction session configuration information for the client. Once the client receives the introduction session configuration message from the DHT, it has

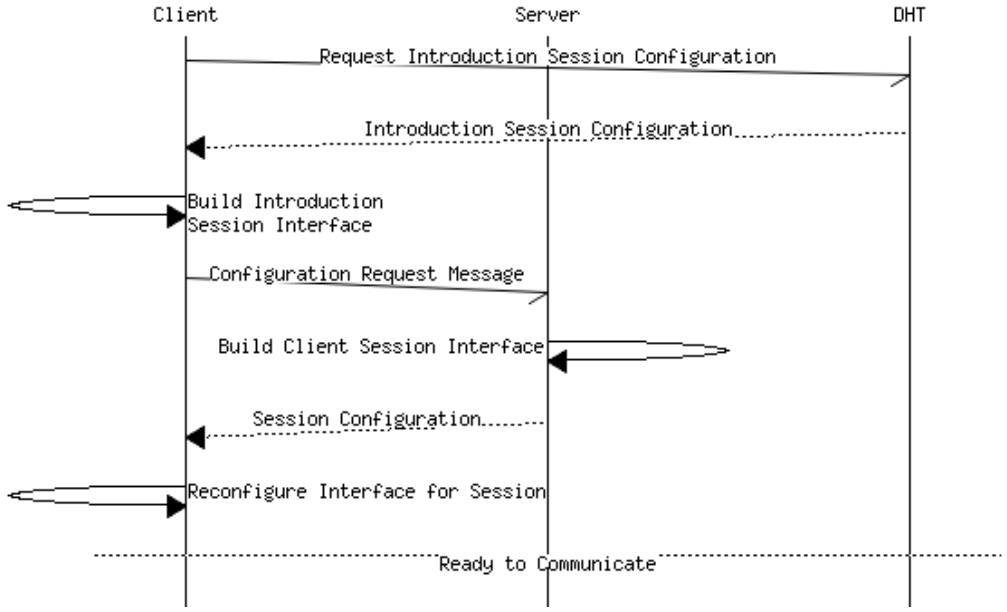

Figure 6.4: Session Establishment Message Sequence Chart

sufficient information to locate the MT6D server on the Internet and begin communication with it.

$$d_T = H(C^i_{pri} \cdot S_{pub})||T)_{0 \rightarrow 159} \tag{6.1}$$

The client's next step is to use the introduction session configuration information in order to instantiate an introduction session interface and begin calculating MT6D addresses for both itself and the server. It then crafts a configuration request message as described earlier in this section and shown in Figure 6.1. The message is wrapped into a standard IPv6 header that is addressed to the server's introduction interface tunnel address which gets routed through the MT6D process where a UDP header and MT6D addressed IPv6 header are added on.

Upon receipt, the server builds a client session interface by instantiating a new tunnel interface, forking a child process that manages the MT6D address calculations for that session interface, and activating an event loop that monitors the interface for any inbound messages. The child process generates a new session key that is

used in the calculation of MT6D addresses for the session and generates a new tunnel interface IPv6 address by concatenating the IPv6 subnet with a randomly generated 64-bit IID. This new address is bound to the session's tunnel interface, which will only be used for communication with a single client for the duration of the session with that client. Once the tunnel interface has been created, an address assigned, and MT6D addresses have begun to be bound to the physical interface, a configuration response message is built and returned to the client. Upon receipt of the configuration response message, the client changes its MT6D session key and modifies its routing tables, ensuring locally originating MT6D traffic is routed into the tunnel that is destined for the server's session interface. The client does not require instantiation of a new MT6D tunnel interface, as the simple modification of the session key is sufficient to begin calculating session MT6D addresses, thus transforming the client's introduction interface into a session interface.

6.2 Results

The proof of concept MT6D server was developed on Debian Linux in C, relying on a combination of *libev* [49], *libsodium* [30], and *libhiredis* [68] to provide event loops, cryptography, and redis server communication respectively. Redis is a simple key/value server that is used as an analog to a full scale distributed hash table. The server was run on a Dell Optiplex 9020 with an Intel i7-4770 CPU running at 3.4GHz and 16GB of RAM. Ten clients were run in a VMware vSphere cluster with access to up to 32 processor cores and 1GB of RAM each. All of the machines were connected to the Virginia Tech production network with 1Gbps network interface cards.

Experiments were conducted in which the time required to transfer a 1GB file from a server to some number of clients over Secure Copy (SCP) was measured. Measurements were taken on experiments that used SCP over TCP and SCP over MT6D. Running experiments with standard SCP over TCP and comparing those results with experiments using SCP over MT6D, clearly shows the overhead introduced by this scheme. Since server performance was being measured, the number

of clients varied from 1 to 10 to demonstrate the feasibility of the server and the session establishment protocol as the number of clients increased. This chapter does not address the theoretical scaling limits of the MT6D server or its protocols, but work to that end has been done and is presented in Chapter 7. Experiments were run over a period of three weeks at a variety of times of day and days of the week in order to demonstrate the little variability that current network conditions have on the server's performance. In order to establish statistical significance, experiments were run until the largest standard error across all varying numbers of clients was less than 0.5 seconds. This resulted in a minimum of 500 data points collected for each number of clients, from 1 to 10.

Figure 6.5 is presented as the first demonstration of the capability of the MT6D server. The plot is a scatter plot of every data point collected throughout all of the experiments. The values along the x-axis represent the varying number of concurrent client connections from 1 to 10, while the y-axis shows the time required to transfer

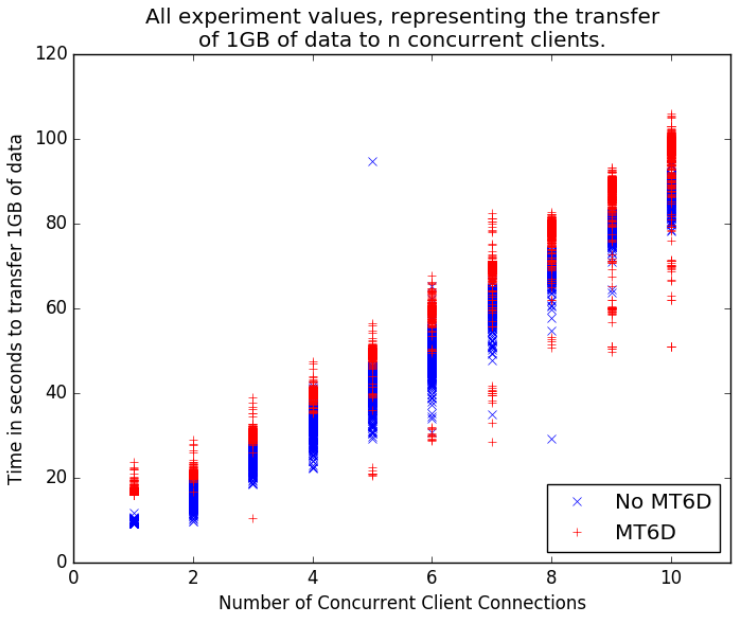

Figure 6.5: A scatter plot showing every data point collected in the transfer 1GB of data to n concurrent clients.

a 1GB file to those clients. This graph shows that there is a steady increase in time required to transfer the target file as the number of clients requesting the file at the same time increases. This graph also demonstrates that with only rare exceptions, using MT6D increases the time required to transfer the file, which is to be expected due to the additional headers that are added on to each packet and thus the reduced amount of data that can be included in a single packet. This simple fact means that more packets will be required to transfer the same amount of data, thus requiring additional resources.

Mean values are calculated on the data presented in Figure 6.5, which results in the line graph shown in Figure 6.6. From this graph, it can be observed that the two lines are approximately parallel, and thus demonstrates that there is an approximately fixed overhead introduced by MT6D as the number of clients increases from 1 to 10. This fixed overhead comes from the fact that MT6D packets are simply IPv6 packets with an additional UDP and IPv6 header added on. By adding a fixed

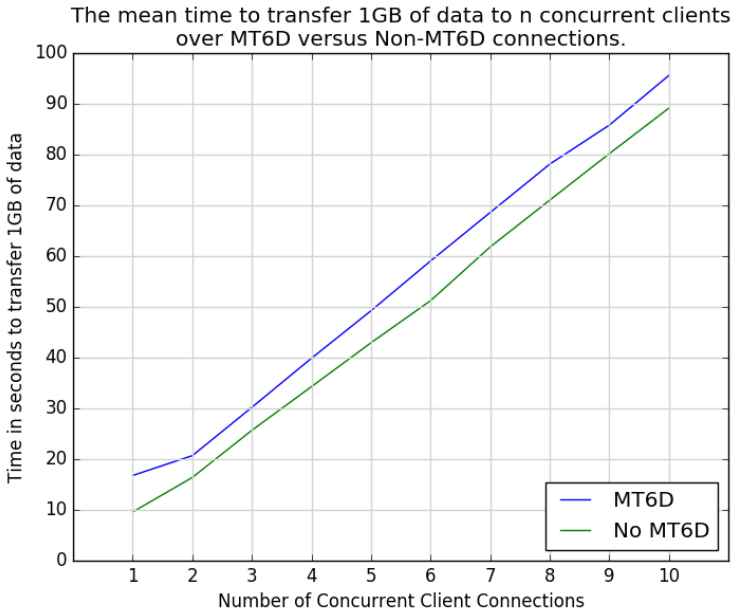

Figure 6.6: The mean time to transfer 1GB of data to n concurrent connections over MT6D versus Non-MT6D connections.

amount of overhead to each packet, the amount of data that must be transmitted is increased in a fixed manner as well.

Finally, Figure 6.7 is presented as a clearer demonstration of the performance impact of using the MT6D server. The graph shows the overhead introduced by MT6D as a percentage of the time to transfer the file without MT6D. More clearly, this graph demonstrates the relationship between the two lines that are presented in Figure 6.6. While there is a slowdown of approximately 75% in the case of 1 client, it must be understood that this represents 7 seconds of overhead for a transfer that takes only 9 seconds to transfer. As the number of concurrent clients is increased, the impact of the increased overhead diminishes and reaches a minimum value of only 7.2% when there are 10 concurrent clients.

Any time an additional layer of security is added to a host or a network, there comes some cost in performance or usability. The presentation of this data demonstrates that the added anonymity, security, and privacy presented by the MT6D

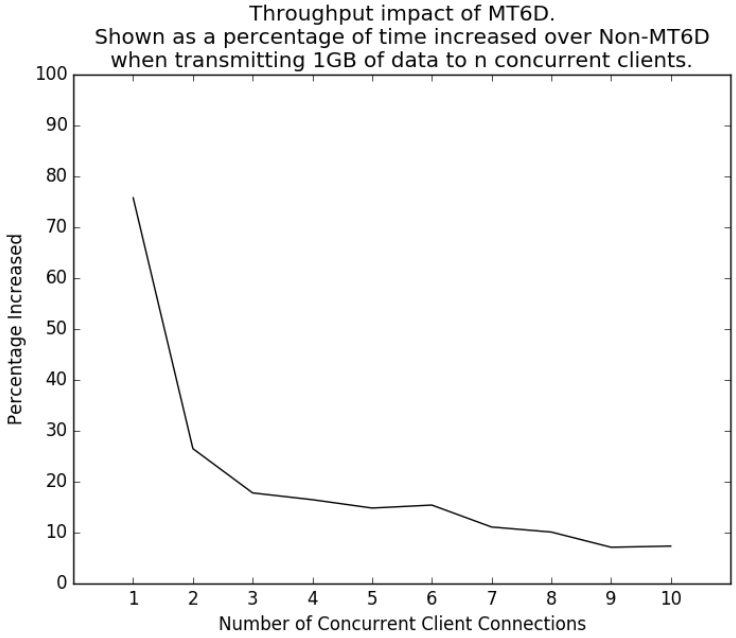

Figure 6.7: Overhead introduced by the MT6D server as the number of concurrent client connections increases.

server comes at a relatively small performance cost.

6.3 Summary

This chapter presented the solution for a server that supports network-based moving target defense. This research has included the design, implementation, and testing of a server that enables multiple clients to connect over independent MT6D connections while sharing a minimal amount of information beforehand. By leveraging the size of the IPv6 address space, this work has improved the security, privacy, and anonymity of both the server and its clients. Additionally, while this work is presented as a solution for client/server networks, many of the features presented could be applied to a peer to peer implementation of MT6D. By nominating one of the peers as a server and understanding that there is only a single client, the scheme is easily used.

The scheme requires only that the server has access to the ECDH generated public key for any of the potential clients that may wish to connect to it. Clients then use their private key and the server's public key in order to collect key configuration information from a distributed hash table where the server had stored it earlier. Maintaining this layer of separation between clients and server provides benefits that include an inability to link the client and server based on standard network communication, which provides a level of anonymity that is not provided by many network services.

Once public keys have been obtained, the scheme provides a means for a client to determine the configuration information required to connect to the server by finding a message that has been left in the DHT by the server. The use of the DHT gives the server the benefit of both network mobility and the ability to modify configuration information periodically. By requiring that the server maintain only a single introduction interface, the amount of computing power required by the server and its network footprint are also reduced. In steady-state, the server maintains only a single set of three MT6D addresses and a single introduction tunnel interface.

Finally, the session establishment protocol is the first time that the server learns of the network location of the client who wishes to establish a connection. Because the server does not maintain any private information about the clients, a compromise of the server's data does not gain an attacker any of the clients' private information. Additionally, the dynamic configuration establishment provides a means through which clients can establish a session from any subnet rather than only those which had been pre-determined. This feature provides a sense of network roaming that is not present in any previous design of MT6D. The only current requirement is that a client remain in the same subnet for the duration of the current session, although with the addition of dynamic re-configuration, a client could potentially begin the reconfiguration process as it transitions from one subnet to another.

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 7

Exploring the Server's Limits

Chapter 6 presented a fully functioning MT6D server as a proof of concept solution to the problem of scaling MT6D up. This chapter explores the theoretical limits of the server in order to grasp the true benefit provided and discover any limitations that may exist. Research presented here focuses on the identification and quantification of the scaling problems that will occur when a computer binds a large number of addresses and permits an equally large number of clients to communicate with the server on those addresses. Ultimately, this research must answer the question of how many clients can actually be supported with a single MT6D server. To this end, it must be determined how many IPv6 addresses can be supported in the modern Linux kernel and how performance is impacted on a server with some large number of addresses bound to it. Additionally, little work has been done to understand the impacts that using MT6D has on a network. This chapter explores those impacts in light of the increased resources required by an MT6D server.

7.1 Server Responsiveness

A number of network services may find some benefit in being obscured with a moving target defense connection. If a user is surreptitiously attempting to share data with someone else via some data transfer protocol, it would be beneficial to

have their location on the network obscured. The user could instead be using Voice over Internet Protocol (VoIP) services and wish to have the location of their phone obscured within the network. While both of these examples could potentially benefit from implementation of a moving target defense, the underlying protocols that support each capability handle network slowdowns much differently. If a user is uploading a file or downloading a webpage, delay in the connection with the server may be acceptable. On the other hand, delay in a real time service such as VoIP has a much greater impact. It is for these reasons that network administrators employ Quality of Service (QoS) tools to ensure that real time service traffic has priority over traffic that is more delay tolerant.

In his book *Usability Engineering* [62], Jakob Nielsen discusses acceptable response times in order to give a user different feelings about the content that they are interacting with. According to Nielsen, responses received within a 0.1 second window make the user feel as though the system is reacting instantaneously. The author states that a one second response time is the generally accepted upper bound for the system to not interfere with a user's flow of thought, although they will notice that there is some delay in the system. These ideas and rough numbers provide inspiration and a desire to keep the server response time as close to the 0.1 second mark as possible. Future sections will show how this helps to dictate the number of addresses and thus the number of clients that a single MT6D server could support.

7.1.1 Methodology

In designing and building an MT6D server in Linux for wide distribution, it is important to understand what is possible given the capabilities provided in the modern Linux kernel using standard commodity hardware. Determining the number of addresses that a machine can bind to its network interface is the first question that must be answered. This helps to determine how many clients are supportable by a single server. Once those addresses are bound, it must also be understood how quickly a server can send and receive data to all of those clients concurrently. This measurement helps to determine the maximum possible rate of data transmission as

the number of active clients changes.

Within Linux, there are a number of different methods that can be used for binding and unbinding IPv6 addresses. The simplest method is a call to the *ifconfig* utility [2] from within the server. This method allows the utility to handle the address binding and unbinding for you without a need to interact directly with the operating system. While simple to implement, this method is extremely inefficient. Each time *ifconfig* is called, a new process must be generated in order to handle the request. Other methods include the sending of *Input Output Control (ioctl)* [3] or *netlink* [6] messages to the kernel. These two methods both reduce the resources required on the machine due to the fact that they communicate directly with the operating system, rather than using a separate utility as a middleman. For the experiments in this section,*netlink* messages were used due to the balance of ease of use and speed.

As there are a number of methods for binding and unbinding addresses, there are also a number of methods that can be used to listen for, receive, process, and respond to client requests on a server. One of the easiest and most common methods is to use standard BSD sockets [9]. Sockets are well supported, but require a reliance on the Linux kernel to process packets from the hardware to the software socket before before being handed over to the server process. Rather than using sockets, it is possible to use zero copy networking [22], where a server that is running in userspace is permitted to reach past the kernel all the way to the network interface card's device driver, reducing the reliance on the kernel's network processing. This method reduces the need for multiple context switches in addition to reducing the number of copies that are required when a packet is moved from the hardware to the server. While zero copy networking is potentially more efficient than standard sockets, it also requires a significantly more complex server. For proof of concept, sockets were used, while leaving zero copy networking for future work. Choosing this method provides a baseline that can be used as a basis for comparison in future work.

7.1.2 Address Bindings

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

These tests were run on a Dell OptiPlex 9020 running 64-bit Debian Wheezy [1]. The machine has an Intel i7-4770 processor running at 3.4Ghz, 16GB of RAM and an Intel I217-LM Gigabit Ethernet network interface card. The experiment consists of a program written in C that uses the *rtnetlink* [6] library to send messages the the kernel for IP Address binding and unbinding. The program begins by generating a list of n IPv6 addresses, where n varies from 1 to 70000 in increments of 5000. The program continues sending *netlink* `RTM_NEWADDR` (bind) messages for each address as quickly as possible. Once all addresses are successfully bound, the program sends `RTM_DELADDR` (unbind) messages as quickly as possible. Address generation, binding, and unbinding were timed individually, and each measurement was averaged across the total number of addresses that were bound and unbound. Each set of parameters was run five times and averaged, resulting in the data that is displayed in Figure 7.1.

The results show that the average time to generate all of the addresses increased linearly as the number of addresses increased. The implementation had a maximum time of two milliseconds to generate 70000 IPv6 addresses as demonstrated by the black line in Figure 7.1 that is barely visible at the bottom of the graph. Experiments found that the average time to generate a single address did not increase, regardless of the number of addresses being generated, and remained consistently between 16 and 48 nanoseconds per address.

The research demonstrates a roughly linear increase in the average time required to bind an address as the number of addresses increased to the 55000 address mark. Above 55000 addresses, an exponential increase in time for each address bound was

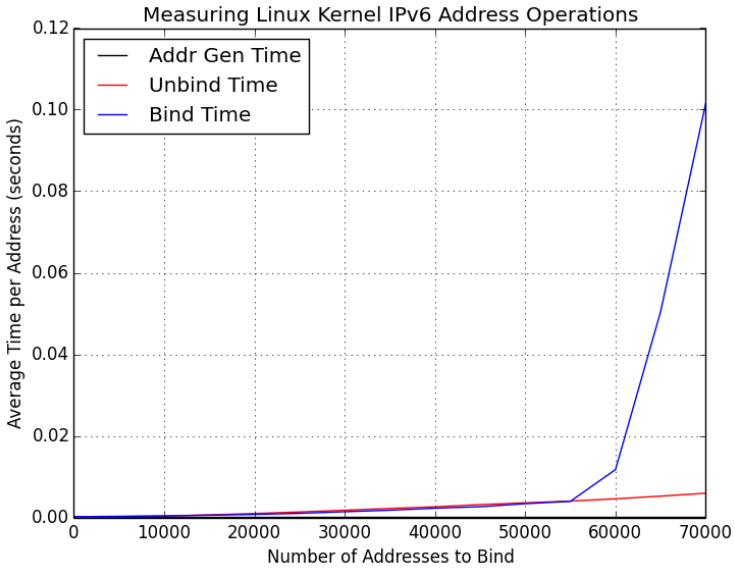

Figure 7.1: The average time required to generate, bind, and unbind an IPv6 address based on the given number of addresses.

discovered, as shown by the blue line in Figure 7.1. The average time to bind addresses increased from 176 μs at one address to 3.9 milliseconds at 50000 addresses. After 55000 addresses, time quickly increases to 11 milliseconds, 50 milliseconds, and 101 milliseconds, for 60000, 65000, and 70000 addresses respectively. The total time required to bind addresses shows a dramatic increase from 216 seconds for 55000 addresses to 7108 seconds for 70000 addresses, which is slowdown of 3100%, while the number of addresses increased by only 27%.

While time to bind has a dramatic increase above 50000 addresses, time to unbind continues to follow a linear slowdown all the way to 70000 addresses. At its slowest, the average time to unbind an address was approximately five milliseconds and is shown by the red line in Figure 7.1.

Basic computer and network functionality was tested after addresses were bound in order to discover whether the limitation was the actual binding of the addresses, or the memory required to keep the addresses bound. Even with 70000 addresses bound to the machine, there was not a notable impact to the performance of the

machine in either basic computing capability or network performance.

Experimental results lead to the conclusion that the current implementation of the MT6D server has an upper bound of approximately 55000 actively bound IPv6 addresses. While it is possible to bind more than 55000 addresses to a machine, the bind time required per address becomes longer than the desired address rotation period, which is approximately three seconds in the default configuration of MT6D. This upper bound also informs the parameters when conducting the experiments described in the next section, which focus on tuning the server for best performance.

7.1.3 Service Performance Metrics

The experiments discussed in the previous section provide a clear understanding of the limits of the server implementation in regards to the maximum number of addresses that can be managed. The next step is to determine the maximum rate of data receipt for the server when bound to that large number of addresses and communicating with that large number of clients. In order to determine these values, a UDP echo server was built that listens on all bound IPv6 addresses on UDP port 3540. When a request is received, the server responds with a quantity of data that is ten times the size of the request. This increase in size is intended to model the standard request/response size ratio that is inherent in most web-based services. This implementation uses standard BSD sockets [9] with *AF_INET6* and *SOCK_DGRAM*.

Once addresses are bound, a listener thread and a worker thread are spawned. The listener thread simply receives UDP requests and queues them in a POSIX message queue [5] as quickly as they are received. The worker thread looks for messages in the queue, and upon receipt of a message, sends the data back to the requesting client ten times. As return packets are built, the program ensures that the source address is the actual address where packet was received, rather than allowing the kernel to address the packet. By default, the current version of the Linux kernel assigns the highest currently bound IPv6 address as the source address unless modified otherwise. Since the worker is sending ten times the data that the listener is receiving, it is clear that the worker thread is the slowest part of the server. Earlier

implementations of the server attempted to leverage multithreading, but contention on the call to BSD sockets *sendmsg()* resulted in much poorer performance.

A production implementation of this server would concurrently communicate with n clients, where n is some number up to 55000. Rather than dealing with the issues that come with managing a group of clients at that scale, traffic generation tools were used to replicate a large scale network. A client traffic generator was built around the Python packet generation tool, Scapy [8], but experimentation showed that the Python interpreter did not generate traffic quickly enough to put any stress on the server. This is primarily to the limitation of using the Python interpreter, rather than compiling code. A new traffic generation tool was created which leverages *libtins* [4]. *libtins* is a C++ library that provides the ease of use of Scapy with the efficiency of compiled C++ code. When the *libtins* client was permitted to send traffic without delay, the traffic generator achieved a rate of approximately 150,000 packets per second (pps), which is approximately 25% of the theoretical limit of a Gigabit Ethernet interface. A sleep timer was added between packet generation iterations to control the rate of transmission in the range of 1900 to 150,000 pps.

In all of the experiments, the server bound n addresses and created a UDP socket in order to listen on all addresses before the client was allowed to proceed. Once the server completed setup and was idly listening on the socket, the client would send 250,000 UDP requests to the server at the dictated rate. Destination IP addresses were selected at random from the pool of addresses that the server had bound. Receipt of fewer than 90% of those 250,000 requests is deemed a failure and those experiment parameters were excluded from the data collection. For this set of experiments, the data rate was varied in the range of 1900 to 150000 pps and the number of addresses in the range from 1 to 55000 in increments of 5000 addresses. The range of packet speed does not fall on clean numbers due to the fact that a sleep timer was controlling packet flow rates. The timer indirectly impacted packet transmission speeds rather than dictating a specific transmission rate. Experiments were run on each set of parameters five times and results were averaged.

Results from these experiments are shown in Figure 7.2. The X axis shows the range of addresses bound to the Ethernet interface from 1 to 55000, while the Y axis

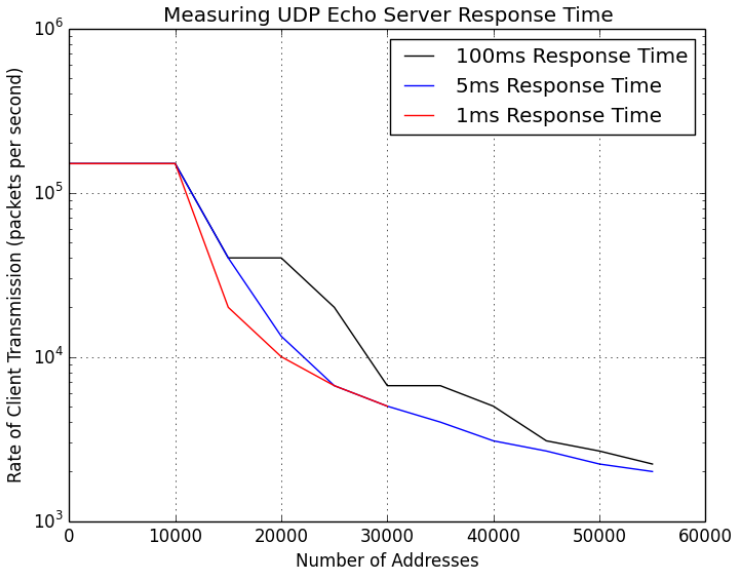

Figure 7.2: The rate of data transmission in packets per second versus the number of addresses bound on the server resulting in the ideal values based on the necessary server response time. The Y axis data is represented using a logarithmic scale.

shows the range of transmission speed from 1900 pps on the bottom to 150000 pps at the top. The data follows a nearly logarithmic function, so the Y axis is presented in a logarithmic scale. The three lines represent the combination of transmission speed and number of addresses bound that result in a successful transmission within the specified response time. As the legend demonstrates, red represents an average response time of less than one millisecond, blue represents an average response time of less than five milliseconds, and black represents an average response time of less than 100 milliseconds.

Results are grouped by average response time due to the fact that different services require a different performance from a server. For example, real time services have much lower tolerance for delay from the server, while other non-real time services such as e-mail are much more tolerant of delays from the server.

No tests with more than 30000 addresses were able to respond in less than one millisecond, no matter the rate of requests. Once a five milliseconds response time

is considered as acceptable, results can be analyzed all the way to 55000 addresses. Unfortunately, there is a maximum possible data rate of approximately 2000 packets per second when the server is bound to 55000 addresses. This means that on average, the server would only be able to support 55000 concurrent clients if their individual data rates averaged one packet every 27 seconds. This is of course fully unacceptable in almost every situation where client/server interaction is required.

As expected, server performance increases as the number of bound addresses decreases, reaching a maximum supportable data rate of 150000 packets per second when 10000 or fewer addresses are bound to the server. When divided among all 10000 of the connected clients the server can sustain an average data rate of approximately 150 packets per second. This of course is still not ideal, but does give a base line from which to build and an understanding of what is possible for an MT6D server. The reality though is that the plateau on the graph when considering fewer than 10000 address is indicative of an inability to fully stress the system. Without a traffic generation system that could push beyond 150000 pps, it is unknown what the performance would be.

Section 3.3 discussed the idea that MT6D keeps three addresses bound to a machine at all times. These addresses are from the previous time period, the current time period, and the next time period. The maintenance of three addresses ensures that two MT6D hosts can continue to communicate regardless of network latency or delays. With this information in hand, there is more context available to understand the true impact of these experiments. Rather than considering only addresses bound, the number of clients with which a server can communicate concurrently is actually what is being measured. By comparing these results with the desired results described at the beginning of Section 7.1, it is apparent that the server should be limited to approximately 10000-15000 concurrent clients. Since MT6D maintains three addresses per client, the server would be limited to 30000-45000 addresses bound at any given time, which falls within the limits presented in Section 7.1.2.

It is also important to consider that not all possible client would be connected to the server at the same time. By understanding the communication duty cycle of the potential clients, further extrapolation could imply that the server could support

some maximum number of clients that is even greater than the 10000-15000 listed above.

7.2 Network Impacts

While MT6D is a proven technology that has been written about and researched extensively, the impact that it has on the network has not been explored in depth. Since MT6D functions by adding additional headers onto normal network data, there is a non-zero impact to the throughput of the applications communicating from within the machine. In addition, each binding and unbinding of an IPv6 address triggers a series of NDP messages to be sent across the subnet. By binding new and unbinding old addresses at each address rotation time period, MT6D is artifically increasing the amount of NDP traffic that is generated, thus increasing the overhead present on the network and decreasing the available bandwidth. This section explores both of these side-effects in order to more clearly understand the impacts of using a network-based moving target defense system such as MT6D on the network.

Simulations were implemented in ns-3 in an effort to clearly model each of the impacts that are described above. Previous efforts to simulate MT6D used the OPNET modeler, but were hindered by OPNET's limited support of IPv6 [25, 26]. Researchers were forced to implement portions of the IPv6 protocol into the modeling tool in order to complete their simulations. ns-3 was selected as the simulation tool for this work due to the full support of the IPv6 protocol that it provides and the fact that ns-3 is open source and thus maintains an extensive following of researchers who contribute to the robustness of the tool.

7.2.1 Impacts of Neighbor Discovery

This section begins by exploring the impact to the network based on the increased NDP generated traffic due to an MT6D server supporting a large number of clients. Determining if NDP messages will flood a device's network queue and cause unsynchronized connections or network delays motivated the creation of ns-3 simulations

to measure how long it takes to bind n addresses and to observe the reaction of NDP as those addresses are bound.

Simulation Setup

This simulations were executed on a Dell Optiplex 990 with an Intel i7 2600 at 3.6GHz with 16GB of RAM. The simulation models a subnet containing a router and a single host. The router node is configured to mimic a Cisco 6509 router and the client node is configured with the attributes of an Intel 82574L Gigabit CT Network Interface Card (NIC), which is one of the most common Gigabit Ethernet cards. In ns-3, the queue size of each node is set as a number of packets rather than a buffer size. The maximum possible buffer size of the NIC in question is 2048kb, which was converted to a packet quantity by calculating the average size of the solicitation messages within the simulation. With a payload length of 32 bytes for each neighbor solicitation message, the buffer grants a queue size of 64000 packets. Besides the router, the client node is the only device within the subnet, and is assigned both link local and global IPv6 addresses. The client and the router are set up using the `IPv6Helper` class provided by ns-3, which allows the nodes to inherit many protocols built into ns-3 including NDP. The nodes within the network are connected via CSMA channels without collision detection through the use of the ns-3's `CsmaHelper` class.

ns-3 provides facilities that enable packet capturing, which provides a record of the simulation in the form of a packet capture, or `pcap` file. `Pcap` files can be examined in any packet analysis tool, such as *tcpdump* or *Wireshark*. They also contain helpful information such as packet content, packet type, source, and destination. In addition, ns-3 provides an `AsciiTraceHelper` class, which permits the study of queue activity during the simulation. The `AsciiTraceHelper` class generates `.tr` files that contain information such as the queuing and whether a packet has been received or dropped. ns-3 also provides network monitors, which permit the observation of simulation command usage for verification purposes. Network monitor classes include `ICMPv6L4Protocol` and `IPv6L3Protocol`. Each time a class calls a function,

the monitor displays the time that the method is called in addition to the parameters and outputs.

Design

The simulation is scripted to attempt to bind n addresses to the client node, where n is one of 100, 500, 1000, 2000, 4000, 8000, 16000, 32000, or 48000. ns-3 automatically assigned IPv6 addresses to the interface through the use of the `IPv6Address`, `IPv6AddressGenerator`, and `IPv6AddressHelper` classes. Figure 7.3 provides a visual representation of the network setup used in the simulation after all addresses are bound. Upon initialization, the router and the client each have only a link local and IPv6 address assigned to them. The two nodes exchange their initial neighbor discovery messages, such as router advertisements and neighbor solicitations immediately after address assignments.

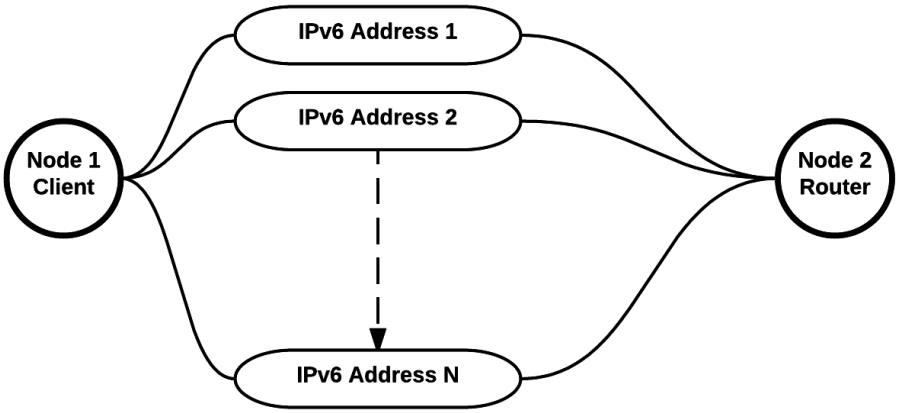

Figure 7.3: Visual representation of the simulated network setup to study Neighbor Discovery Protocol.

At 100 seconds of simulation time, ns-3 strips the client device of its IPv6 address and begins to bind new and unique IPv6 addresses. New address binding is delayed by 100 seconds to ensure that all initialization of the subnet is complete and the network is stable. Because this study focuses on the impact on NDP of binding

a large number of addresses, the address used does not matter, so incrementation is used for address generation rather than fully implementing the MT6D address generation scheme. Additionally, by incrementing addresses rather than calculating them, address collision avoidance is guaranteed.

After binding addresses, the client node sends a neighbor solicitation message for each address bound. After the message is transmitted, the router receives and handles them according to ns-3 functions implemented in the `ICMPv6L4Protocol` class. The total time required to send all NS messages is then recorded after the successful receipt of the final NDP message. To also measure the influence of bandwidth on this problem, each varying value of n is simulated with connection speeds of 10Mbps, 100Mbps, 1Gbps, 10Gbps, and 100Gbps.

Results

Table 7.1 shows the results for 16000 and 32000 addresses. The simulation demonstrates that at speeds of 100Mbps, 1Gbps, 10Gbps, and 100Gbps address bindings occur within 0.5% of each other when binding the same number of addresses. On a network of 10Mbps, the results show that there is very little impact to time. This same trend was also observed in all other simulations with different numbers of addresses. The difference in binding time between the varying connection speeds remained proportionally the same.

Once it was determined that bandwidth has little impact on the time to bind addresses, experiments were limited to using only a connection speed of 1Gbps.

Table 7.1: Table of recorded times for 16000 and 32000 addresses. Time indicates the time required to successfully bind all addresses.

Bandwidth	16000 Addresses (Seconds)	32000 Addresses (Seconds)
10 Mbps	38.406	76.883
100 Mbps	37.384	74.799
1 Gbps	37.266	74.546
10 Gbps	37.255	74.54
100 Gbps	37.253	74.538

This speed was chosen due to the prevalence of 1Gbps connections on commodity hardware. The simulation results confirm that there is a direct impact between the number of addresses and the time it takes to send all ICMPv6 NDP messages at each address binding. Figure 7.4 shows the recorded time for each varying number of addresses. The results indicate that the address bindings can be completed in as little as 0.232 seconds for 100 addresses. On the other extreme, it takes 112.273 seconds to bind 48000 addresses to the client. The time required to generate and handle these neighbor solicitation messages appears to be trivial, however the time required to transmit and receive the messages appears to slow down the address binding tremendously. The client node requires a fixed amount of time to push a solicitation message onto the queue, send the message, and pop the solicitation message out of the queue. The rate at which the host sent all ICMPv6 messages

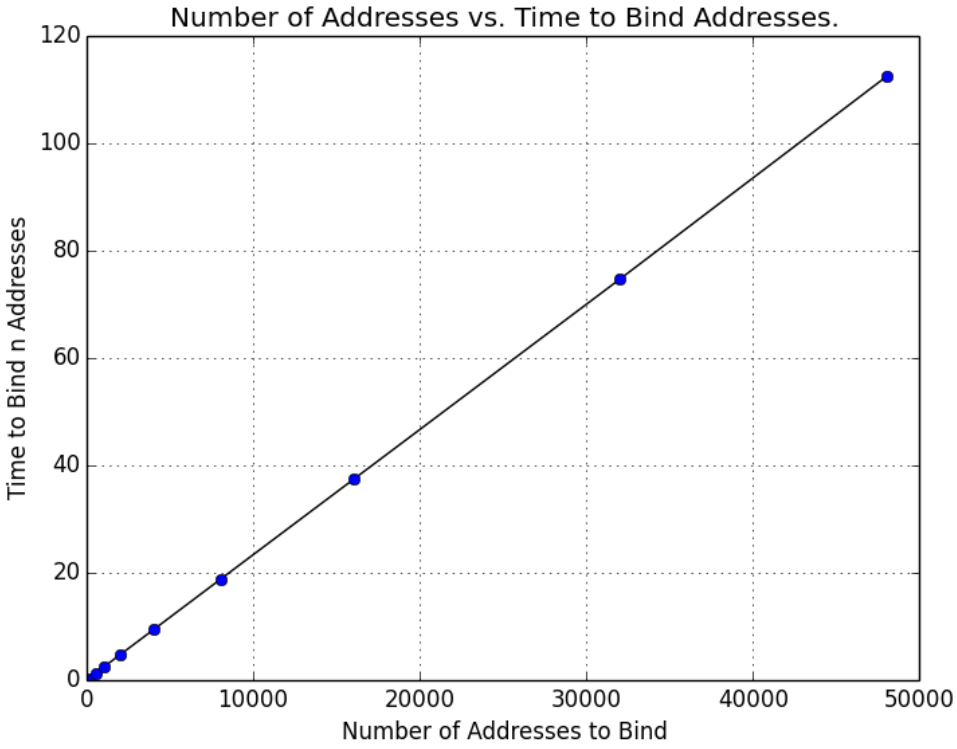

Figure 7.4: The time required to bind n addresses successfully.

grew linearly with the increase in address number. A scalar multiplication on the two previously mentioned results yields an approximation of all other results with less than 0.5% error. This linear correlation is also evident in the trend line exhibited in Figure 7.4.

According to the simulations, it can take up to 2.34 milliseconds to completely bind a single IPv6 address. Using this information and applying it to the linear correlation between the number of addresses and the time required to bind those addresses, a linear equation is discovered that produces an approximate address change time for a varying number of clients. Approximate address change time can be calculated using $f(n) = 0.00234(n * 3)$, where n represents the number of MT6D clients. For example, the expected address change interval for 15000 addresses is 35.1 seconds. This is the fastest time that guarantees all addresses are successfully bound before the next scheduled address change should occur. However, this does not take account for any other data transfer or communication, therefore the user may want to allocate more time to allow other traffic to pass.

7.2.2 Impacts of Increased Overhead

One aspect of the MT6D method of conducting network-based moving target defense that needs to be addressed is the measurement of overhead incurred by encapsulating packets in an MT6D tunnel. When a two MT6D hosts connect through an MT6D tunnel, traffic sent through the tunnel is encapsulated inside of a UDP datagram and an additional MT6D addressed IPv6 packet. The addition of these two layers incurs a total of 48 bytes of overhead, 40 bytes for the IPv6 header and eight bytes for the UDP header. Determine how this additional overhead impacts large data transfers through MT6D is important.

When sending data across a network, a series of fragmentation and encapsulation steps are required according to the protocols in use. In the case of this research, the focus is on TCP over IPv6 on an Ethernet network segment. In order to simplify the problem slightly, a fixed TCP window size that is matched to the Ethernet Maximum Transmission Unit (MTU) is assumed. Upon transmission, data is fragmented into

pieces that will fit into each size protocol data unit as the data is passed down the protocol stack. In the case of TCP/IPv6 over Ethernet, data must be broken into pieces no larger than 1440 bytes. 1440 comes from the 1500-byte MTU of Ethernet, with room reserved for a 40-byte IPv6 header and a 20-byte TCP header. Once these frames, packets, and segments traverse their portion of the network, they are reassembled into the data as it originally existed at the sender.

In order to measure the impact the 48-byte MT6D overhead has on large file transfers, the number of packets required to send one gigabyte of data through the standard network listed above is calculated and compare that to the number of packets required to send the same data through an MT6D network. For these calculations, a file of exactly 1GB is transferred from one host to another. 1GB of data is equal to 1,073,741,824 bytes. Since each packet will become fragmented according to the MTU of the network, the data will be broken into 745,655 1440-byte Ethernet frames. Once the additional overhead of MT6D is added into the equation, the MTU of the network is reduced by 48 bytes due to the addition of a 40-byte MT6D addressed IPv6 header and an 8-byte UDP header. When transmitted over MT6D, the 1GB file is now broken into 771,367 1392-byte Ethernet frames, which is an increase of 25,712 Ethernet frames. This increase to traffic on the network only considers the case in which a single client is transmitting.

To put this increase of traffic into terms of time, it is important to know that at the maximum transmission speed of a 1Gbps Ethernet interface, it takes approximately 1.526 μs to transmit a single frame. This means that the addition of the MT6D overhead brings an increase of approximately 37 milliseconds to transmit a 1GB file. Of course it is understood that faster or slower interfaces will impact this number significantly, as will network conditions in which maximum transmission speeds are not achievable. Finally, based on the work discussed in 7.1, there is a theoretical limit of 15,000 clients that can be connected concurrently to an MT6D server. In this upper limit case, there is a slowdown of 565 seconds across all of the clients. This slowdown is minimally impactful, but should be understood by someone who desires to implement MT6D as a solution in their network.

There are networks that are configured to support jumbo Ethernet frames, which

significantly decrease the impact of MT6D introduced overhead. Jumbo frames increase the Ethernet MTU from 1500 bytes to 9000 bytes. Once space is reserved in the frame for IPv6 and TCP headers, the non-MT6D MTU becomes 8940 bytes and the MT6D MTU becomes 8892 bytes. The 1GB file now only requires 120,106 frames without MT6D and 120,754 with MT6D for a difference of only 648 frames. This increases transmission time by 988 μs in the case of a single client transmitting and 14.8 seconds if the theoretical limit of 15,000 concurrent clients could be reached. While this is minimally impactful, jumbo frames are not widely supported across the Internet, and in fact the 9000-byte MTU is a widely used example, but is not defined in any standard.

7.2.3 Address and Port Collisions

Section 3.3 presented some discussion on the probability of an address collision occurring. Equation 7.1 enables the calculation of the probability of an address collision within a subnet that has h active hosts.

$$P_c = \frac{h}{2^{64}} \tag{7.1}$$

According to Equation 7.1, there is only a $5.42 \times 10^{-15}\%$ chance of a collision on a subnet with 100,000 active addresses. While this number is extremely small, there must be some understanding of the impact to the protocol should an address collision occur. Any address collision would be detected by the NDP DAD process that occurs each time an IPv6 address is bound to an interface. Should a duplicate address be detected, the calculated address would not be bound. Since addresses are calculated by the endpoints of a session, there is no facility that permits a node to notify its peer of a collision. This means that for the duration of the current address rotation time period, the colliding host would be unable to receive any MT6D traffic.

In addition to calculating IPv6 addresses, MT6D also has the option of calculating UDP ports as well. A UDP port is a 16-bit integer that is used by a host to determine which application on the machine the inbound traffic should be redirected to. Since there are only 2^{16} or 65,536 ports available, the likelihood of a port collision is fairly

high. At the upper limits of 15,000 clients, the server is maintaining a total of 45,003 IPv6 addresses. By modifying Equation 7.1, the probability of collision is calculated as:

$$P_c = \frac{h}{2^{16}} = \frac{45002}{2^{16}} = 68.6\% \tag{7.2}$$

While there is a fairly high probability that port collisions will occur, this is minimally impactful and potentially beneficial. As discussed in Section 3.3, MT6D generates port numbers in order to prevent correlation between packets and the potential reconstruction of network flows, thus defeating the benefits of MT6D. Since the MT6D server supports a large number of clients, an adversary would see an extremely large number of ports being consumed. Port collisions could lead the adversary to assume that common ports are used by communicating pairs, which could lead them to attempt to reassemble network flows. Of course this is not the case, and could help to additionally confuse the attackers attempts to understand the traffic passing between MT6D hosts.

Port collisions provide no concern for implementation or functioning of the server either. The modern Linux kernel, upon which the MT6D server proof of concept was built, permits multiple processes to listen on common port numbers. The server uses a combination of source and destination IPv6 addresses and source and destination UDP port numbers to determine which client a packet arrives from. By allowing the MT6D server to listen on all calculated ports and filtering traffic based on all available header information, it is certain that the server can separate communication sessions between all of its connected clients.

7.3 Summary

While Chapter 6 presented a fully functional MT6D server, the limits of the server must be understood. This chapter explored not only limitations that stem from the physical hardware that the server is running on, but also limitations from the network on which the server resides. Based on all of the research presented

in this chapter, each server that is running on hardware equivalent to the testing environment should be limited to supporting approximately 10,000-15,000 clients. By pushing beyond those limitations, the implementer could cause excessive delays in either packet processing or transmission.

This research has exposed some restrictions of developing an MT6D server following the current IPv6 architecture. Understanding the limitations of MT6D client/server communications will not only provide guidelines and insight into the use of such systems, but will also enable further studies into the engineering of solutions for these limitations. Demonstrating the NDP bottleneck in an MT6D server, motivates the development of a solution which could permit an MT6D server to match the efficiency of traditional single address servers. Lastly, the use and application of this research is not necessarily limited to the scope of MT6D. Understanding the relation between servers, routers, and bandwidth is essential to improving client server architecture-based systems. Further study of NDP can also help to improve efficiency for other systems and provide new incentives to migrate to IPv6.

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 8

Visualizing a Moving Target Defense

The old saying goes, "A picture is worth a thousand words." This is especially true when trying to understand the complexity of large scale networking problems and protocols. According to the author of *Visualization Viewpoints*,

> The amount of information available to scientists from large-scale simulations, experiments, and data collection is unprecedented. In many instances, the abundance and variety of information can be overwhelming [47].

The author also discusses the scientific importance of abstracting data in such a way that it is meaningful to the human brain. In the case of IPv6 and network-based moving target defenses, the human brain is unable to fully grasp the massive size of 2^{128} addresses. Analogies are attempted, such as comparing the volume of a one meter sphere as the IPv4 address space and the volume of three times the sun as the IPv6 address space, but the human mind is still unable to truly comprehend the size of these numbers. In order to overcome this limitation, a simple website visualization tool was built that helps to see and understand what is happening not only in the IPv6 address space, but also with MT6D and the impact that it has on data traversing a network.

8.1 MT6D Server Demonstration System Architecture

The implementation of this visualization tool begins with the addition of Node.JS [37] web servers running on each of the MT6D machines. The primary web server is installed on the same machine as the MT6D server. Web servers installed with the MT6D clients are only used for the receipt of http POST messages in order to trigger some action in the MT6D client. Node.JS was selected due to a combination of its lightweight nature and full support of Web 2.0 technologies. In addition to Node.JS, express [45], socket.io [70], and shelljs [83] modules were used. Express helps with writing quicker, cleaner, and easier to read javascript code. Socket.io allows the creation of a web socket between the web server and web client, thus allowing the server to push data to the client without requiring the client to continually poll the server for new data. Shelljs gives Node.JS the ability to run specifically selected shell scripts on the web server.

Figure 8.1 provides a visual diagram to more easily understand the architecture of the demonstration system. Red lines denote DHT configuration exchange com-

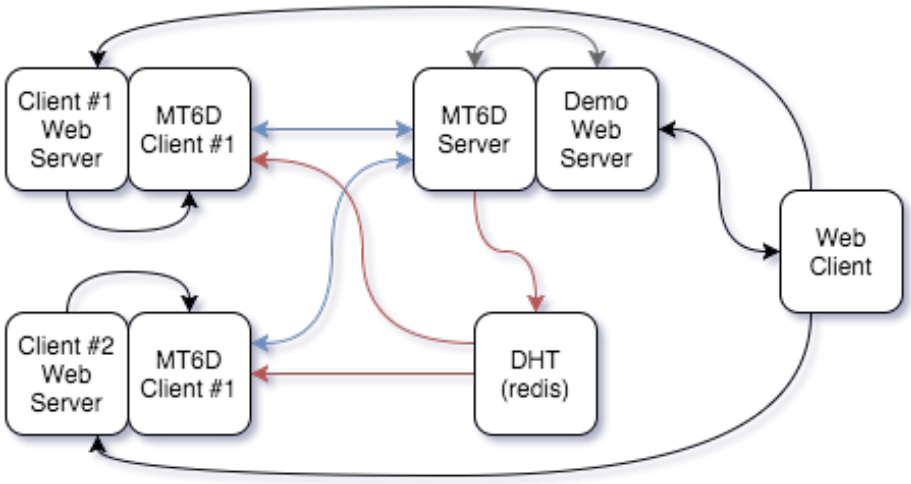

Figure 8.1: The MT6D server visualization system architecture.

munication paths. Note that the configuration is generated on the MT6D server, pushed into the DHT, and retrieved from the DHT by the MT6D clients. Blue lines denote MT6D session establishment and MT6D communication paths. With configurations retrieved from the DHT, the clients have the ability to establish sessions directly with the MT6D server. Black lines denote web traffic. All demonstration triggers are sent from the web client to either the demonstration web server or one of the client web servers. Upon receipt, web servers use shelljs to execute MT6D on their hardware. The version of MT6D that is running on the MT6D server has been customized to send data to the primary web server via http POST. This data includes address calculations and the status of ongoing connections with each of its respective clients. The data is then relayed to the web client through a web socket in the form of small JSON objects. The web client is then able to display the data in a way that is meaningful to the user.

8.2 Watching Addresses Change

The first part of the visualization is to gain an understanding of the address space. As can be seen in the left portion of Figure 8.2, the local 64-bit IPv6 subnet is represented in two dimensions. In this representation, the X-axis represents the first 32-bits of the IID and the last 32-bits of the IID are represented on the Y-axis. Each axis consists a total of 2^{32} or approximately 4 billion addresses. On a 27 inch monitor, these 4 billion addresses fit into approximately 4 inches or one billion addresses per inch. At this scale, the cross-sectional diameter of 0.5mm mechanical pencil lead covers approximately 20 million addresses or an area of approximately 100 trillion addresses. This is still an extremely large number, but is at least a number that people have heard of. As addresses are bound to the MT6D machines, the plot on the visualization tool is updated to show both the introduction interface address and the addresses of connections between the server and its clients. Each color represents a different pair of addresses. One address on the client's session interface on the server and the other address on the client. The lone black dot is

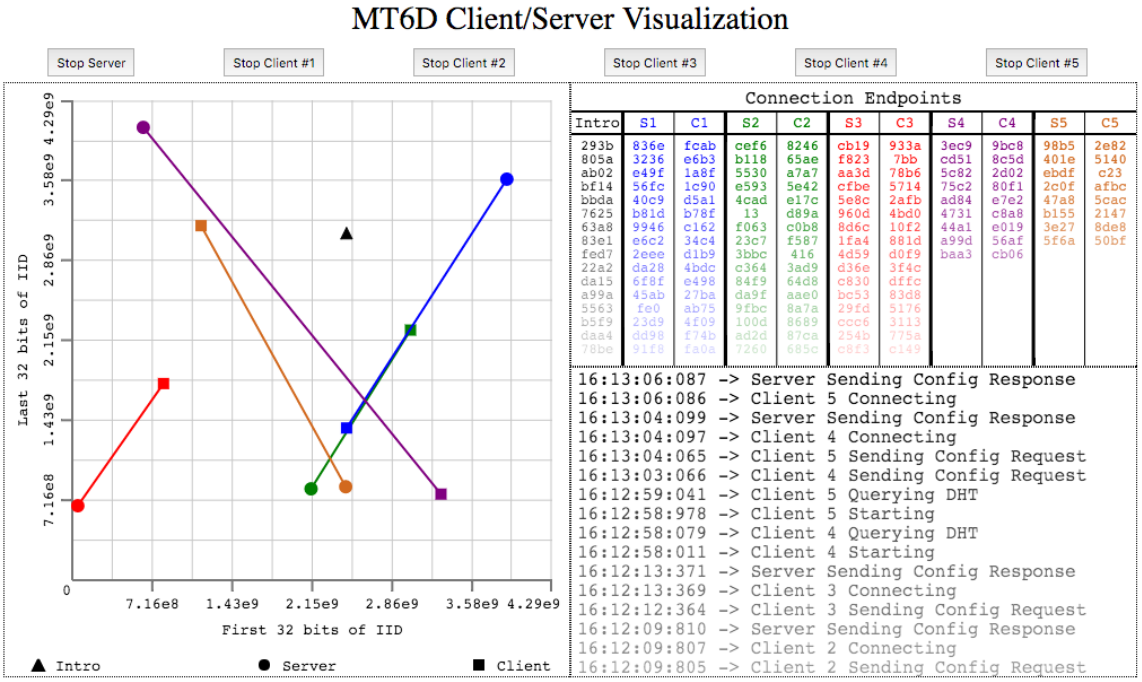

Figure 8.2: Visualizing the power of the MT6D address hopping scheme.

the introduction address at which the DHT configuration information will lead the clients in order to begin the session establishment process.

In the top right corner of Figure 8.2, a user can see the addresses of each of the MT6D machines as they change in addition the the most recent expired addresses. In order to make the display less busy, only the last 16 bits of the IPv6 address is shown. Note also that the color of the addresses in the display matches the color of the line representing that specific address pair. In the bottom right corner of the visualization, the user can learn the current status of each of the machines involved in the demonstration in addition to understanding the flow of messages during the DHT configuration exchange process and the session establishment process.

When using this visualization to demonstrate the power of MT6D, people are often challenged to use a mechanical pencil to predict the exact location of the next circle in the 2 dimensional plot. If they are able to predict the location, they are then

reminded that the tip of that pencil covers approximately 100 trillion addresses and the machine occupying that address will only be there for a short time. This method of demonstrating MT6D helps people who don't truly have an understanding of what is happening to see how difficult it is to find a computer that is using MT6D.

8.3 MT6D's Impact on Streaming Video

Beyond understanding what is happening in MT6D, some people believe that the utilization of the scheme will slow network performance in some noticeable manner. The final portion of the visualization website shows a comparison of streaming video from each of the clients as can be seen in Figure 8.3. The video is streamed from webcams through the MT6D clients to the MT6D server over an MT6D session. For comparison purposes, the visualization shows an MT6D routed video on the left and a video streamed directly to the browser on the right. By comparing the streaming rate of each of these videos, an observer can see that MT6D has no noticable impact. While this demonstration may not provide much information in print form, it is easy to understand that MT6D has little impact to network performance when viewed on a computer.

8.4 Summary

MT6D is a complex topic that many people have difficulty fully grasping. It is with this in mind that this visualization came to fruition. By providing a tool like that presented in this chapter, the technology created through this research becomes accessible to a much larger audience. Rather than focusing on audiences with the technical understanding, this visualization is built for those that may not understand the details of the scheme, but still wish to understand the high level concepts.

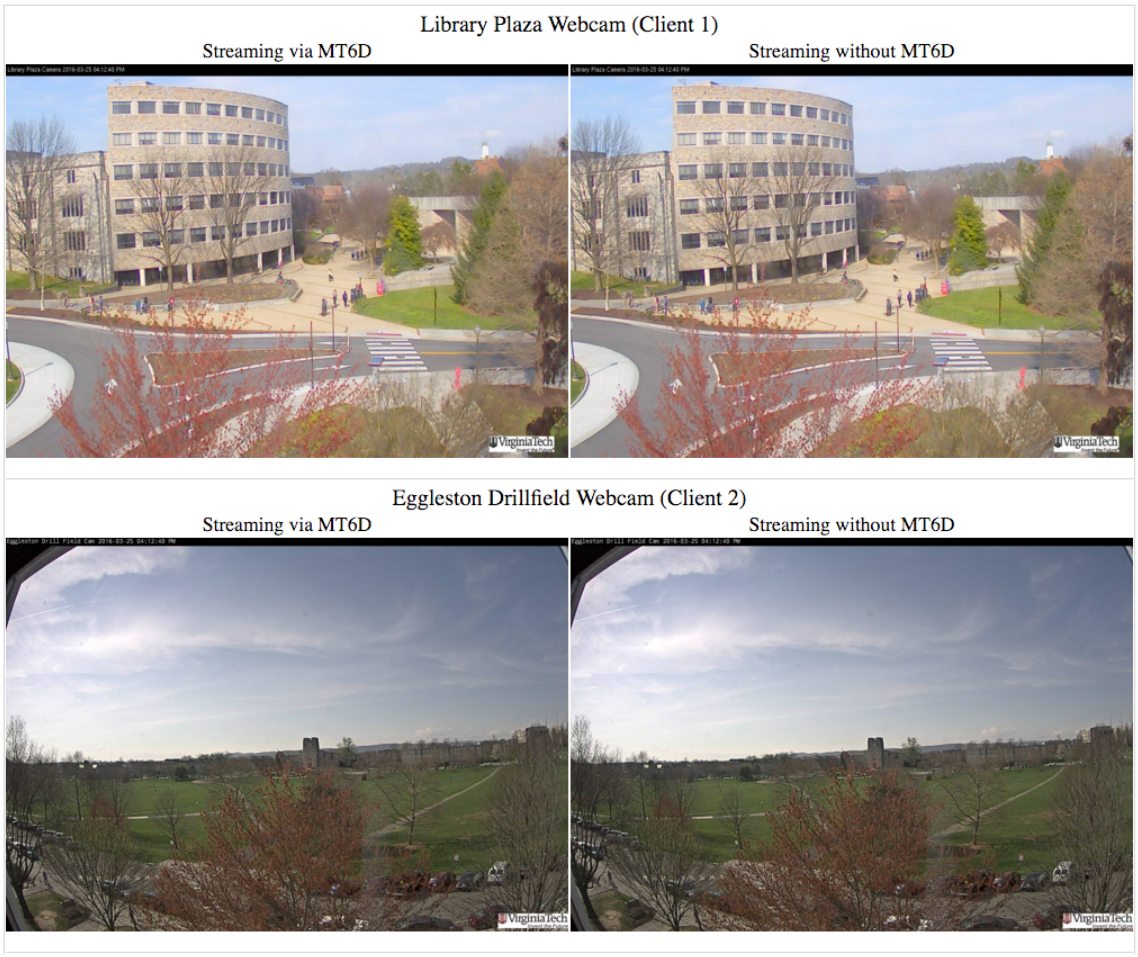

Figure 8.3: Demonstrating the performance of video streaming over an MT6D connection.

Chapter 9

Conclusion

This chapter concludes the dissertation by presenting a summary of the research that was conducted as part of this work. Additionally, it will discuss the contributions of the research and provide a discussion of some of the limitations of the MT6D server. Finally, this chapter will provide some direction to those who which to continue this line of research and will close with some concluding thoughts.

9.1 Summary of Research

As described in Section 1.1, "The purpose of this research effort is to design, implement, and test a technique that extends a network layer moving target defense to support a client/server network." This dissertation has provided the building blocks required to create this server and culminated in a functional proof of concept server that could theoretically support up to 15,000 clients. The chapters here provide the pieces that were assembled in the conduct of this work.

Chapter 1 introduced the problem, providing motivation for the research and an understanding of the threat that is being defended against. Chapter 1 also established the objectives of this research and provided a list of questions that provided guidance as the research moved forward.

Chapter 2 is a literature review that provides background on a number of different

applications of network-based moving target defenses. In particular, it provides descriptions of similar technologies to which this research can be compared.

This work is based on the combination of a number of different technologies which were the focus of Chapter 3. Most importantly, Chapter 3 provided the requisite background to understand the state of MT6D before this research began.

Chapter 4 presented a novel technique that was developed in conjunction with Reese Moore that permitted the exchange of arbitrary data through the use of a DHT as a form of escrow. The application of public/private key cryptography enabled a data exchange method that provided a level of anonymity to those unaware of what was happening, but at the same time provide the authenticity and security necessary for exchanging key data.

Building upon the blind rendezvous techniques introduced in Chapter 4, a scheme was developed that allows MT6D hosts to exchange requisite configuration information anonymously and securely. Presented in Chapter 5, this scheme resolves many of the limitations present in the original implementation of MT6D, paving the way for the creation of an MT6D server.

Chapter 6 is the core of the paper that combines everything presented in the first 5 chapters into the design, implementation, and testing of a fully functional MT6D server. The server presented answers the research questions asked in Chapter 1 and meets the ultimate goals of this research.

While the design and implementation of the server was core to this research, additional analysis was necessary to understand the limitations of the technology. Chapter 7 presented work that analyzed the impact of the physical hardware and operation system on the performance of the MT6D server in addition to the impact of the server on the local subnet in which it resides. This testing included a combination of implementation and measurement, simulation, and theoretical analysis.

Chapter 8 provides a brief aside from the core of the research to demonstrate the power that visualizing a technology such as MT6D has on the ability to understand it. Chapter 8 presented a visualization tool that provides a way for people to more easily understand MT6D in addition to seeing the impact of the technology on the transfer of data between two machines.

Finally, appendix A provides the code that was created in the implementation of this server.

9.2 Contributions

This dissertation makes three primary contributions to the field of network-based moving target defense. First, Chapter 4 presents a means to conduct exchange information in a way that provides anonymity, authentication, and security. Second, Chapter 5 presents a technique that gives MT6D the capability to exchange configuration information by only sharing public keys. Finally, Chapter 6 presents the MT6D session establishment protocol.

DHT Blind Rendezvous

The research establishes a novel technique through which the exchange of information can be conducted while remaining anonymous to outside observers yet still authenticated to a peer. While the combination of anonymous and authenticated seems counter-intuitive, the scheme manages to succeed at this, which provides a new capability to a number of privacy preserving technologies. As discussed in Chapter 4, there is a BitTorrent Extension Proposal that would provide the ability to store some arbitrary 1000 bytes of data within the DHT, although that is only a small piece of the problem that is solved here. The key contribution presented here is the use of private/public key pairs in conjunction with a time element in order to generate a time sensitive shared location at which some key data could be shared.

Dynamic Configuration Exchange

MT6D was limited in functionality due to its reliance on statically defined configuration information. The lack of dynamic configuration exchange prevented the ability for peers to move around the network and prevented any growth beyond a small handful of peers communicating with each other. Dynamic configuration

exchange solves these problems, providing the ability for two peers to exchange configuration information and establish an MT6D connection. Additionally, none of the technologies presented in Chapter 2 attempt to conduct a blind rendezvous. Each scheme presented relies on some out-of-band configuration to take place before any connection could be established. It is this ability to conduct a blind rendezvous that facilitates the scaling and mobility required of a client/server relationship.

Session Establishment Protocol

Once MT6D had the ability to dynamically exchange configuration data, it was possible for two clients to establish an MT6D connection using the exchanged data. However, this configuration exchange capability did not provide the means to establish a large number of connections concurrently as would be necessary for a server. The creation of the Session Establishment Protocol extends dynamic configuration exchange from focusing on simple configuration to becoming the first step in the establishment of a session. Rather than including all of the information required to establish a session between two peers, dynamic configuration exchange provides only the means for clients to find their server. With the server's location now known to the clients, the clients use the session establishment protocol to dynamically request the generation of a session specific MT6D configuration. This protocol is the key to the successful extension of MT6D from a peer to peer technology to something that is capable of supporting many concurrent users.

9.3 Limitations

As with the original implementation of MT6D, it must be understood that there are weaknesses present with this research that must be fully considered in order to gain the most benefit. Most importantly, no layer 3 address changing scheme can provide protection from an attacker that has unfettered access to the local subnet. Due to the way that the TCP/IP/Ethernet protocol stack functions and the fact that every Internet Protocol (IP) packet is encapsulated inside of an Ethernet frame.

Regardless of the address that is in the IP packet header, the MAC address of the Ethernet NIC is in every frame, which points back directly to the sending machine. It is for this reason that the local subnet must be included in the TCB as discussed in Section 1.3.

Chapter 7 addressed in detail the capacity limitations of the MT6D server presented in this work. Based on that research, a single MT6D server should be limited to supporting fewer than 15,000 clients. While this is a hard limit that was discovered in the course of this work, it is based on a specific implementation on specific hardware. There is the possibility that these limits could be increased by using higher performance hardware and continuing with research directions that are presented in the next section.

9.4 Future Work

Through the course of conducing this research, many interesting paths were discovered that were outside of the scope of this work. These paths could include additional testing, improved performance, or features that were not implemented in the proof of concept MT6D server presented in this work. The intent of this section is to explore some of these thoughts for the benefit of other researchers who may choose to continue this work.

BitTorrent DHT

As was described in chapters 4 and 5, this research stopped short of conducting configuration exchanges through the actual BitTorrent DHT and rather chose to use Redis as a DHT analog. BitTorrent Enhancement Proposal (BEP) 44 [64] provides a technique that could be leveraged to push this portion of the research forward. Also, the work presented here suggests the BitTorrent Mainline DHT as the preferred solution for configuration exchange, but this need not be the only solution. There are other DHT's functioning on the Internet today that could be used instead. Rather than use a DHT, an implementer could potentially develop some other means of

exchanging configuration information. In order to gain the benefits provided through this research, the selected medium should be large enough to provide a level of anonymity and reliability that is gained from the Mainline DHT. It could even be possible that there is an implementation where a simple key/value server such as Redis would meet the needs of an implementer.

Additional Testing

Chapter 6 demonstrated the ability of the server to support up to 10 clients, while Chapter 7 showed that the theoretical limit of the server is likely in the range of 10000-15000 clients. When testing the actual server, focus was placed on demonstrating that the server can actually communicate with multiple clients concurrently rather than demonstrating any limits. The testing presented in chapters 6 and 7 could be combined in order to test the true limits of the server and supporting protocols. This level of testing would require the creation and coordination of 10,000 to 15,000 clients, which is far beyond the scope of the work presented here, although it could provide a great deal of interesting data and potential improvements to this work. Researchers may consider the use of Linux LXC containers, which function as a form of lightweight virtual machine, as a means of scaling up to a very large number of clients.

In regards to the key exchange mechanism, benchmarking should be done in order to demonstrate how easily the mechanism can scale, and what effect it has on the efficiency of the node publishing key exchange information to the DHT for hundreds or thousands of potential clients. Should this large scale DHT publication be determined as a limiting factor, there is a potential mechanism that could be leveraged. In order to allow the number of clients to scale past this bottleneck, a co-opt of the two-stage message publications from the DP5 privacy preserving presence protocol [21] could be used. Instead of publishing a new DHT message for each client during each window, the scheme could publish a short term DHT key for each client over some long window and use that key to publish a single introduction message to the DHT on the shorter window. For n clients and a long-term key length of w

DHT publication windows, this reduces the number of DHT messages from nw to $n + w$, potentially saving a large amount of network traffic and other computational resources as n grows. This technique would the execution of an additional query by each client, but that work would be spread across all clients while the amount of work place on the server would decrease significantly.

Increasing Capacity

During the course of conducting the work presented in Chapter 7, it was discovered that there are two primary factors that limit the performance of an MT6D server. These factors are the time required to bind an address and the time required to receive, process, and reply to a UDP request. The methods selected to execute these tasks within the operating system were convenient, but perhaps not the most efficient methods possible. For example, there are several methods that can be used to bind IP addresses to an interface. Two methods were analyzed, focusing on using either the *ifconfig* utility or sending *netlink* messages. As reported, there was a great performance increase by moving from the former to the latter. There is a third method that involves sending *ioctl* messages, which do not rely on an additional library as *netlink* messages do. It is possible that transitioning from one to the other could bring some performance increases. Additionally, there is the possibility that the server could bypass the utilities that are provided by the operating system and reach directly into the data structures that hold IP addresses and manipulate them manually. There is the potential for great risk in using this method, but more research could be conducted to determine if it is feasible.

While decreasing address binding times is important, a greater issue is that of sending and receiving data to a large number of clients using different IPv6 addrsses concurrently. In this proof of concept implementation, standard BSD sockets with *SOCK_DGRAM* UDP sockets were used to send and receive data between the machines on the network. This is not the most efficient means of passing data onto a network, but it is the easiest and most well supported. Research could be conducted to determine if the use of raw sockets [9], zero copy networking [22, 75], or

the PF_Ring framework [7] could improve server efficiency. Raw sockets allow the creation of a network datagram manually rather than relying on the Linux kernel. It is possible that code could be created that creates the IPv6 and Ethernet headers more efficiently than the kernel, thus resulting in faster data processing. Zero copy networking is a method in which the ring buffer that is normally used by the kernel's network stack for send and receive queues is moved from kernel space into user space. This simple technique eliminates the need for the server to copy data from kernel space to user space on receipt or to copy data from user space to kernel space on send. This technique can also be implemented in such a way that system calls are nearly eliminated, thus resulting in far fewer time consuming context switches. PF_Ring is a framework that exploits zero copy networking and implements a large portion of the low level code required to make zero copy networking function, thus abstracting much of the problem away from the server implementer.

Adding Features

The session management protocol introduced in Chapter 6 defines three message types that are used for the establishment and tear down of MT6D sessions. There is the possibility that adding a fourth message type to the protocol could improve efficiency and reliability. As the protocol is currently described, configuration request messages and configuration response messages are triggers that set a timer in the host that sends them. If that timer expires before a response is received by the server, the session will be terminated. In the case of a client, timer expiration triggers a new attempt at sending a configuration request. By adding an acknowledgement message into the protocol, hosts would gain the knowledge of what is happening on the other end of the conversation much more quickly. The use of an acknowledgement would not improve the reliability of the protocol, but could help to slightly improve the efficiency of the machines involved since they would not have to wait for a timer to expire in order to take the next step forward in the conversation with their peer.

There is ongoing research in which researchers are attempting to modify the MT6D address generation algorithm to function in low powered IoT or hardware de-

vices. Current research suggests that the key limiting factor on low powered hardware is the inability to execute the complex task of hashing with the SHA256 hashing algorithm as is standard in MT6D. If an alternate hashing algorithm were implemented in these low powered client devices, the server would also have to be modified in order to support them. At the same time, there is the possibility that there could be clients that wish to continue using SHA256 for the hashing of addresses. The server could be extended to support a number of hashing algorithms, using the messages exchanged during the session establishment process to find a common hashing algorithm, much the way SSL and TLS agree on encryption algorithms during session establishment. The risk that this presents is that the server's introduction interface address calculations would have to be executed by using the simplest hashing algorithm that any of the clients may potentially use. An alternate solution to this scenario is that MT6D address calculations could be conducted at the edge of the IoT wireless network in much the same way as the network based moving target defense schemes presented in Chapter 2 are conducted. This would permit addresses to be calculated on hardware that is much more powerful than traditional IoT sensor devices, while still retaining many of the benefits that the MT6D scheme provides.

Finally, while the scheme meets the definition of dynamic configuration, there remains the possibility of increasing the dynamism of the solution by allowing the modification of configuration information mid-session. As the scheme is currently defined, configuration information is determined upon connection, but then remains constant for the duration of a connection. If one of the endpoints of the session should move to a new subnet, the protocol would have no way to update the configuration without tearing down and re-building the session. By adding several new messages, one of the hosts could gain the ability to request a re-configuration on the fly. The system would require some trigger for this action, a means to notify the peer, and some way to return a configuration, but the act of updating the configuration itself would be a relatively simple addition to the current scheme. Mobile IPv6 as presented in RFC 3775 [48] and RFC 6275 [67] could provide some inspiration for a solution to this problem somewhat similar to the work presented by Heydari [43].

9.5 Concluding Thoughts

This dissertation has presented a successful solution to the problem of extending MT6D in support of a client/server network. In fact, it is the only known instance of a purpose built network-based moving target defense server designed to support a large number of clients. Chapter 4 demonstrated a means through which the BitTorrent DHT could be used as a means of exchanging arbitrary data while maintaining a level of anonymity and privacy. Chapter 5 explored the application of DHT blind rendezvous techniques to MT6D in order to enable scalable and dynamic configuration exchanges with the need to only exchange public keys before connecting. Chapter 6 presented the design, implementation, and testing of a functional MT6D proof of concept. Chapter 7 discussed the theoretical limits of the MT6D server from both the perspective of the actual server and the perspective of the network on which the server is functioning.

While the proof of concept implementation presented here is limited in functionality, it does function and provides the techniques required to continue to push the state of the art forward. Network-based moving target defense systems are not in the mainstream yet, as there are a significant number of issues to overcome in order to make them easy to deploy, use, and maintain. The improvements to MT6D that have been presented in this research bring researchers one step closer to making moving target defense a reality across the Internet, which will then allow them to leverage the size of the IPv6 address space to improve the security, privacy, and anonymity of both servers and their respective clients. Additionally, while this work was presented as a solution for client/server networks, many of the features presented could be applied to a peer to peer implementation of MT6D. By nominating one of the peers as a server and understanding that there is only a single client, the scheme is easily used.

To bring back the analogy presented at the beginning of Chapter 1, I have successfully designed, built, and tested a castle that maintains its normal defense in depth protections, is capable of moving to arbitrary locations at will, yet continues

to maintain services for all those who desire to consume them. Through this work, I hope that I have made a small impact on the security posture of the Internet at large.

THIS PAGE INTENTIONALLY LEFT BLANK.

Bibliography

[1] Debian wheezy homepage. https://www.debian.org/ports/amd64/. [Online: accessed 15 September 2014].

[2] ifconfig - linux man page. http://man7.org/linux/man-pages/man8/ifconfig.8.html. [Online; accessed 15 September 2014].

[3] ioctl - linux man page. http://man7.org/linux/man-pages/man2/ioctl.2.html. [Online; accessed 15 September 2014].

[4] Libtins packet crafting and sniffing library. http://libtins.github.io/. [Online; accessed 15 September 2014].

[5] mqoverview - linux man page. http://man7.org/linux/man-pages/man7/mq_overview.7.html. [Online: accessed 15 September 2014].

[6] netlink. http://man7.org/linux/man-pages/man7/netlink.7.html. [Online; accessed 4 June 2014].

[7] Pf_ring. http://www.ntop.org/products/pf_ring/pf_ring-zc-zero-copy/. [Online; accessed 4 June 2014].

[8] Scapy. http://www.secdev.org/projects/scapy/. [Online; accessed 15 September 2014].

[9] socket. http://man7.org/linux/man-pages/man2/socket.2.html. [Online; accessed 8 March 2014].

[10] The role of dns in botnet command & control. Technical report, OpenDNS, 2012.

[11] Ehab Al-Shaer. Toward network configuration randomization for moving target defense. In *Moving Target Defense*, pages 153–159. Springer, 2011.

[12] Spyros Antonatos, Periklis Akritidis, Evangelos P Markatos, and Kostas G Anagnostakis. Defending against hitlist worms using network address space randomization. *Computer Networks*, 51(12):3471–3490, 2007.

[13] J. Arkko, V. Devarapalli, and F. Dupont. Using IPsec to Protect Mobile IPv6 Signaling Between Mobile Nodes and Home Agents. RFC 3776 (Proposed Standard), June 2004. Updated by RFC 4877.

[14] Dileep Basam, Randy Marchany, and Joseph G Tront. Attention: moving target defense networks, how well are you moving? In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 54. ACM, 2015.

[15] Dileep Kumar Basam. *Strengthening MT6D Defenses with Darknet and Honeypot capabilities*. PhD thesis, Virginia Tech, 2015.

[16] Daniel J Bernstein. The poly1305-aes message-authentication code, 2005.

[17] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[18] Daniel J Bernstein. Extending the salsa20 nonce. In *Workshop Record of Symmetric Key Encryption Workshop 2011*, 2011.

[19] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[20] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. Updated by RFCs 5246, 7027.

[21] Nikita Borisov, George Danezis, and Ian Goldberg. Dp5: A private presence service. Technical report, Technical Report 2014-10, Centre for Applied Cryptographic Research (CACR), University of Waterloo, 2014.

[22] J.S. Chase, AJ. Gallatin, and K.G. Yocum. End system optimizations for high-speed tcp. *Communications Magazine, IEEE*, 39(4):68–74, Apr 2001.

[23] A Chavez, Jason Hamlet, Erik Lee, Mitchell Martin, and William Stout. Network randomization and dynamic defense for critical infrastructure systems. *Sandia National Laboratories Report–SAND2015-3324*, 2015.

[24] Hyunsang Choi, Hanwoo Lee, Heejo Lee, and Hyogon Kim. Botnet detection by monitoring group activities in dns traffic. In *Computer and Information Technology, 2007. CIT 2007. 7th IEEE International Conference on*, pages 715–720. IEEE, 2007.

[25] Brittany Clore, Matthew Dunlop, Randy Marchany, and Joseph Tront. Validating a custom ipv6 security application using opnet modeler. In *MILITARY COMMUNICATIONS CONFERENCE, 2012-MILCOM 2012*, pages 1–6. IEEE, 2012.

[26] Brittany Michelle Clore. *Evaluating Standard and Custom Applications in IPv6 Within a Simulation Framework*. PhD thesis, Virginia Polytechnic Institute and State University, 2012.

[27] B Cohen. Bep 3: The bittorrent protocol specification, jan. 2008, 2008.

[28] Bram Cohen. The bittorrent protocol specification, 2008.

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

[30] Frank (jedisct1) Denis. libsodium. `https://github.com/jedisct1/libsodium`. [Online; accessed 10 February 2015].

[31] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[32] R. Droms. Dynamic Host Configuration Protocol. RFC 1541 (Proposed Standard), October 1993. Obsoleted by RFC 2131.

[33] Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. MT6D: a moving target IPv6 defense. In *MILITARY COMMUNICATIONS CONFERENCE, 2011-MILCOM 2011*, pages 1321–1326, 2011.

[34] Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. The blind man's bluff approach to security using IPv6. *Security & Privacy, IEEE*, 10(4):35–43, 2012.

[35] Anthony Ephremides, Jeffrey E Wieselthier, and Dennis J Baker. A design concept for reliable mobile radio networks with frequency hopping signaling. *Proceedings of the IEEE*, 75(1):56–73, 1987.

[36] Bruce A Fette. *Cognitive radio technology*. Academic Press, 2009.

[37] The Linux Foundation. Node.js. `https://nodejs.org/`. [Online; accessed 20 July 2015].

[38] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), August 2011.

[39] Marc Green, Douglas C MacFarland, Doran R Smestad, and Craig A Shue. Characterizing network-based moving target defenses. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 31–35. ACM, 2015.

[40] Stephen Groat, Matthew Dunlop, Randy Marchany, and Joseph Tront. The privacy implications of stateless IPv6 addressing. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 52:1–52:4, New York, NY, USA, 2010. ACM.

[41] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. 2008.

[42] Zhaoquan Gu, Qiang-Sheng Hua, Yuexuan Wang, and Francis Lau. Nearly optimal asynchronous blind rendezvous algorithm for cognitive radio networks. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on*, pages 371–379. IEEE, 2013.

[43] Vahid Heydari and Seong-Moo Yoo. Moving target defense enhanced by mobile ipv6.

[44] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 2373 (Proposed Standard), July 1998. Obsoleted by RFC 3513.

[45] TJ Holowaychuk. Express. `http://expressjs.com/`. [Online; accessed 22 February 2016].

[46] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132. ACM, 2012.

[47] Chris Johnson. Top scientific visualization research problems. *Computer graphics and applications, IEEE*, 24(4):13–17, 2004.

[48] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775 (Proposed Standard), June 2004. Obsoleted by RFC 6275.

[49] Marc Lehmann. libev. `http://linux.die.net/man/3/ev`. [Online; accessed 4 September 2014].

[50] Andrew Loewenstern. Bittorrent dht protocol. *BitTorrent BEP*, 5, 2008.

[51] Douglas C MacFarland and Craig A Shue. The sdn shuffle: creating a moving-target defense using host-based software-defined networking. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 37–41. ACM, 2015.

[52] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[53] John Michalski, Carrie Price, Stanton Stanton, Erik Lee, Yip Heng Wong, and Chung Pheng TAN. Network security mechanisms utilizing dynamic network address translation. 2002.

[54] Robert L. Mitchell. As ipv4 addresses run low, fears of ip cybersquatting increase. *ComputerWorld*, 2010.

[55] Jens Mittag. Dsn research group-live monitoring. *History*, 9:06, 2014.

[56] Reese Moore, Christopher Morrell, Randy Marchany, and Joseph G Tront. Utilizing the BitTorrent DHT for blind rendezvous and information exchange. In *Milcom 2015 Track 3 - Cyber Security and Trusted Computing (Milcom 2015 Track 3)*, Tampa, USA, October 2015.

[57] Christopher Morrell, Reese Moore, Randy Marchany, and Joseph G Tront. Dht blind rendezvous for session establishment in network layer moving target defenses. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 77–84. ACM, 2015.

[58] T. Narten and R. Draves. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 3041 (Proposed Standard), January 2001. Obsoleted by RFC 4941.

[59] T. Narten, R. Draves, and S. Krishnan. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941 (Draft Standard), September 2007.

[60] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC 2461 (Draft Standard), December 1998. Obsoleted by RFC 4861, updated by RFC 4311.

[61] 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.

[62] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[63] A Norberg. Bep 42: Dht security extension, 2014.

[64] Arvid Norberg and Steven Siloti. Bep 44: Storing arbitrary data in the dht, 2014.

[65] Hamed Okhravi, MA Rabe, TJ Mayberry, WG Leonard, TR Hobson, D Bigelow, and WW Streilein. Survey of cyber moving target techniques. Technical report, DTIC Document, 2013.

[66] Guillermo Owen and Gordon H McCormick. Finding a moving fugitive. a game theoretic representation of search. *Computers & Operations Research*, 35(6):1944–1962, 2008.

[67] C. Perkins, D. Johnson, and J. Arkko. Mobility Support in IPv6. RFC 6275 (Proposed Standard), July 2011.

[68] Pivotal. redis. `http://redis.io/`. [Online; accessed 10 February 2015].

[69] Tanner Preiss, Matthew Sherburne, Randy Marchany, and Joseph Tront. Implementing dynamic address changes in contikios. In *Information Society (i-Society), 2014 International Conference on*, pages 222–227. IEEE, 2014.

[70] Guillermo Rauch. socket.io. `http://socket.io/`. [Online; accessed 22 February 2016].

[71] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996. Updated by RFC 6761.

[72] Andrew Robertson, Lan Tran, Joseph Molnar, and Er-Hsien Frank Fu. Experimental comparison of blind rendezvous algorithms for tactical networks. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012 IEEE International Symposium on a*, pages 1–6. IEEE, 2012.

[73] Matthew Sherburne, Randy Marchany, and Joseph Tront. Implementing moving target ipv6 defense to secure 6lowpan in the internet of things and smart grid. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 37–40. ACM, 2014.

[74] Matthew Gilbert Sherburne. *Micro-Moving Target IPv6 Defense for 6LoWPAN and the Internet of Things*. PhD thesis, Virginia Polytechnic Institute and State University, 2015.

[75] Jia Song and Jim Alves-Foss. Performance review of zero copy techniques. *International Journal of Computer Science and Security (IJCSS)*, 6(4):256, 2012.

[76] Mario Strasser, Christina Popper, Srdjan Capkun, and Mario Cagalj. Jamming-resistant key establishment using uncoordinated frequency hopping. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 64–78. IEEE, 2008.

[77] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC 2462 (Draft Standard), December 1998. Obsoleted by RFC 4862.

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

[79] Joseph D Touch, Gregory G Finn, Yu-Shun Wang, and Lars Eggert. Dynabone: dynamic defense using multi-layer internet overlays. In *null*, page 271. IEEE, 2003.

[80] Liang Wang and J. Kangasharju. Measuring large-scale distributed systems: case of bittorrent mainline dht. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10, Sept 2013.

[81] Justin Yackoski, Jason Li, Scott A. DeLoach, and Xinming Ou. Mission-oriented moving target defense based on cryptographically strong network dynamics. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, CSIIRW '13, pages 57:1–57:4, New York, NY, USA, 2013. ACM.

[82] Kara Zaffarano, Joshua Taylor, and Samuel Hamilton. A quantitative framework for moving target defense effectiveness evaluation. In *Proceedings of the Second ACM Workshop on Moving Target Defense*, pages 3–10. ACM, 2015.

[83] Nicholas Zakas. shelljs. `https://www.npmjs.com/package/shelljs-nodecli`. [Online; accessed 22 February 2016].

[84] Shuyuan Zhang, Sharad Malik, Sanjai Narain, and Laurent Vanbever. In-band update for network routing policy migration. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 356–361. IEEE, 2014.

[85] Rui Zhuang, Scott A DeLoach, and Xinming Ou. A model for analyzing the effect of moving target defenses on enterprise networks. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 73–76. ACM, 2014.

[86] Rui Zhuang, Scott A DeLoach, and Xinming Ou. Towards a theory of moving target defense. In *Proceedings of the First ACM Workshop on Moving Target Defense*, pages 31–40. ACM, 2014.

[87] Rui Zhuang, Su Zhang, Alex Bardas, Scott A. DeLoach, Xinming Ou, and Anoop Singhal. Investigating the application of moving target defenses to network security. In *2013 6th International Symposium on Resilient Control Systems (ISRCS)*, pages 162–169, 2013.

[88] Rui Zhuang, Su Zhang, Scott A. DeLoach, Xinming Ou, and Anoop Singhal. Simulation-based approaches to studying effectiveness of moving-target network defense.

Appendix A

Proof-Of-Concept Implementation

This appendix provides an overview of the proof-of-concept implementation created in support of this research effort. While the entirety of the code is not included here, the provided information should convey an understanding of the implementation structure. The entire codebase can be obtained by contacting Dr. Joseph G. Tront (jgtront@vt.edu), Mr. Randolph C. Marchany (marchany@vt.edu), or Dr. Christopher Morrell (morrell@vt.edu).

A.1 mt6d.h

mt6d.h is the primary header file for the proof-of-concept implementation. It consists primarily of the global macros, data structure definitions, and global function prototypes.

A.1.1 Macros

A number of macros are included in mt6d.h to establish default values for a number of variables throughout the code. This list of macros is included in Code Sample A.1.

Code Sample A.1: Proof-of-Concept Global Macros

```
#define ADDR_ROT_TIME 2
#define CONF_ROT_TIME 10
#define DEFAULT_CLIENT_ID 0
#define MT6D_KEY_LEN 16
#define MT6D_ADDRS 3
#define MAX_CLIENTS 10
#define MAX_PKT_SZ 4096
#define MAX_IDLE_COUNTER 10
```

A.1.2 MT6D Association Structure

The MT6D association structure shown in Code Sample A.2 is the data structure that contains all of the data required by a running server or client. In the case of the server, a child process is generated for each individual session. Each child process maintains its own MT6D association structure, reducing the level of inter-process communication required.

Code Sample A.2: The MT6D association structure

```
struct mt6d_assoc {
   struct {
      struct in6_addr base_src;
      struct in6_addr base_dst[MAX_CLIENTS];
      uint8_t         key[MT6D_KEY_LEN];
      int             rot_secs;
      int             interface;
      int      server_flag;
      int      conn_info_rot_secs;
      int      numClients;
   } conf;

   struct {
      unsigned char my_public[crypto_box_PUBLICKEYBYTES];
      unsigned char my_secret[crypto_box_SECRETKEYBYTES];
      unsigned char
         peer_public[MAX_CLIENTS][crypto_box_PUBLICKEYBYTES];
```

```
   } keys;

   struct ev_loop *loop;
   ev_signal      sigw;
   ev_periodic    rotw;
   ev_periodic    confrotw;
   ev_periodic    clientw;
   ev_io tunw;
   ev_io diew;

   int      mypipefd[MAX_CLIENTS][2];
   int      clientid;
   int      myclientid;
   int   mypid;
   struct in6_addr tunsrcaddr;
   struct in6_addr tundstaddr;
   int      tuniface;
   int      tunfd;
   int      last_contact_counter;
   struct {
      struct in6_addr src;
      struct in6_addr dst;
      uint16_t srcport;
      uint16_t dstport;
      int             sock;
      ev_io           w;
   } bound_addrs[MT6D_ADDRS];
   int             curaddr;
};
```

A.1.3 Global Function Prototypes

Code sample A.3 provides a listing of the global function prototypes for the proof-of-concept implementation.

Code Sample A.3: Global function prototypes

```
void mt6d_udp_pkt_cb(EV_P_ ev_io *w, int revents);
void mt6d_tun_pkt_cb(EV_P_ ev_io *w, int revents);
void mt6d_addr_rotate_cb(EV_P_ ev_periodic *w, int revents);
void mt6d_config_rotate_cb(EV_P_ ev_periodic *w, int revents);
void mt6d_init_loop(struct ev_loop*, int argc, char *argv[]);
void mt6d_cleanup(struct ev_loop *, struct ev_signal *, int);
void send_dying_message(struct mt6d_assoc *mt6d);
void mt6d_unbind_all(struct mt6d_assoc *mt6d);
void gen_session_key(uint8_t *newKey);
int iface_up(int iface);
int route_add(int iface, struct in6_addr *addr, uint32_t prefix);
int route_del(int iface, struct in6_addr *addr, uint32_t prefix);
int iface_bind_ipv6(int iface, struct in6_addr *ip);
int iface_unbind_ipv6(int iface, struct in6_addr *ip);
int tun_alloc(char *dev, struct mt6d_assoc *mt6d);
int send_msg(struct mt6d_assoc *mt6d, unsigned char *payload, int
   payload_sz);
int mt6d_client_init_cb(EV_P_ ev_periodic *w, int revents);
int build_new_session(struct mt6d_assoc *mt6d, struct
   config_request *request);
int join_new_session(struct mt6d_assoc *mt6d, struct
   config_response *reponse);
void mt6d_init_conf(struct mt6d_assoc *mt6d, int argc, char
   *argv[]);
void mt6d_hash(struct mt6d_assoc*, struct in6_addr*, uint16_t*);
void make_sockaddr(struct sockaddr_in6 *ssa, struct in6_addr
   *saddr, uint16_t port);
```

A.2 Included Libraries

The proof-of-concept implementation relies heavily on functions provided by *libev*, *libsodium*, and *libhiredis*.

A.2.1 *libev*

libev is an event loop library that provides file descriptor watchers and timers used throughout the implementation. File descriptor watchers are used to maintain awareness of outbound packets arriving on the tunnel interface or inbound packets arriving on the physical interface. These watchers trigger the MT6D transformation, either encrypting and encapsulating an outbound packet or decapsulating and decrypting an inbound packet. Setting up a file descriptor requires the three lines of code demonstrated in Code Sample A.4.

Code Sample A.4: Establishing a file descriptor watcher with *libev*.

```
mt6d->tunw.data = mt6d;
ev_io_init(&mt6d->tunw, mt6d_tun_pkt_cb, mt6d->tunfd, EV_READ);
ev_io_start(loop, &mt6d->tunw);
```

File descriptor watchers are also used to provide inter-process communication. Before a process exits, it sends a message via an inter-process pipe that is being watched by the other processes. This allows all of the processes to maintain awareness of the state of other processes within the system.

libev timers are used to maintain the address rotation timer and the configuration rotation timer. Address rotation timers are used by both the client and the server to trigger new address calculations and bindings. The configuration rotation timer is only used by the server and triggers the generation of a new introduction session key and publication of that key and other required configuration information to the DHT. Setting up a timer requires the code shown in Code Sample A.5.

Code Sample A.5: Establishing a *libev* timer.

```
mt6d->rotw.data = mt6d;
ev_periodic_init(&mt6d->rotw, mt6d_addr_rotate_cb, 0.,
   mt6d->conf.rot_secs, 0);
ev_periodic_start(loop, &mt6d->rotw);
```

In addition to the above cases, *libev* is also used to watch for `SIGINT` in order to exit MT6D gracefully. In the case of a client, a `SIGINT` triggers the sending of a session teardown message to the server. It also ensures that addresses are cleanly

unbound from the physical interface and memory is cleaned up before exiting. In the case of the server, a `SIGINT` triggers the sending of session teardown messages to all of the currently connected clients in addition to the cleanup of addresses bound to the physical interface and any necessary memory cleanup as well.

A.2.2 *libsodium*

The Sodium crypto library provides an easy to use library for a large variety of cryptographic functions. In the proof-of-concept implementation for this work, *libsodium* provides all key generation, hashing, encryption, and decryption functions. Code snippets for each of these examples are provided as Code Samples A.6, A.7, A.8, and A.9.

Code Sample A.6: Generating ECDH keys using *libsodium*.

```
unsigned char server_sk[crypto_box_SECRETKEYBYTES] = {0};
unsigned char server_pk[crypto_box_PUBLICKEYBYTES] = {0};

crypto_box_keypair(server_pk,server_sk);
FILE *serverKeyStore = fopen("./keystore/serverkeys.key","w");

size_t sz;

sz = fwrite(server_pk, crypto_box_PUBLICKEYBYTES, 1,
   serverKeyStore);
if (sz == 0) {
   printf("fwrite error\n");
}
sz = fwrite(server_sk, crypto_box_SECRETKEYBYTES, 1,
   serverKeyStore);

unsigned char client_sk[crypto_box_SECRETKEYBYTES] = {0};
unsigned char client_pk[crypto_box_PUBLICKEYBYTES] = {0};
FILE *clientKeyStore;
char buffer[20];
int i;
```

```
   for (i = 0; i < numClients; i++) {

      printf("Generating Client %d Key Pair\n",i);
      crypto_box_keypair(client_pk,client_sk);
      sprintf(buffer,"./keystore/client%dkeys.key",i);
      clientKeyStore = fopen(buffer,"w");

      sz = fwrite(client_pk, crypto_box_PUBLICKEYBYTES, 1,
         clientKeyStore);
      sz = fwrite(client_sk, crypto_box_SECRETKEYBYTES, 1,
         clientKeyStore);
      sz = fwrite(server_pk, crypto_box_PUBLICKEYBYTES, 1,
         clientKeyStore);
      sz = fwrite(client_pk, crypto_box_PUBLICKEYBYTES, 1,
         serverKeyStore);
      fclose(clientKeyStore);
   }

   fclose(serverKeyStore);
```

Code Sample A.7: Hashing MT6D addresses using *libsodium*.

```
void mt6d_hash(struct mt6d_assoc *mt6d, struct in6_addr *addr,
   uint16_t *port) {
   unsigned char hash[crypto_hash_sha256_BYTES] = {0};
   crypto_hash_sha256_state state;

   /* Calculate the time_part */
   uint32_t t = time_part(mt6d->conf.rot_secs, mt6d->loop);

   /* Perform the hash in parts */
   crypto_hash_sha256_init(&state);
   crypto_hash_sha256_update(&state, &(addr->s6_addr[8]), 8);
   crypto_hash_sha256_update(&state, mt6d->conf.key, MT6D_KEY_LEN);
   crypto_hash_sha256_update(&state, (unsigned char*)&t,
      sizeof(uint32_t));
   crypto_hash_sha256_final(&state, hash);
```

```
    memcpy(&(addr->s6_addr[8]), hash, 8);
    *port = ((uint16_t*)hash)[4];
}
```

Code Sample A.8: Conducting public key encryption using *libsodium*.

```
unsigned char nonce[crypto_box_NONCEBYTES];
randombytes_buf(nonce, sizeof nonce);
unsigned char encmsg[DHTMSGLEN + 1] = {0};
if (crypto_box_easy(encmsg + crypto_box_NONCEBYTES,
          (unsigned char *)raw_msg,
          sizeof(struct kexmsg),
          nonce,
          client_publickey,
          server_secretkey) < 0) {
    printf("Failed to Encrypt.\n");
    exit(EXIT_FAILURE);
}
```

Code Sample A.9: Conducting public key decryption using *libsodium*.

```
const unsigned char *nonce = binmsg;
return crypto_box_open_easy((unsigned char *)kexmsg,
       binmsg + crypto_box_NONCEBYTES,
       sizeof(struct kexmsg) + crypto_box_MACBYTES,
       nonce,
       server_publickey, client_secretkey);
```

A.2.3 *libhireds*

Redis is an in-memory data structure store that is run for this research as a key/value server analog to the BitTorrent Mainline DHT. *Redis* and the C library *hiredis* provide an easy to use Application Programming Interface (API) that is demonstrated in Code Samples A.10 and A.11.

Code Sample A.10: Publishing a key/value pair to the DHT analog using *libhireds*.

```
while (valuematch != 0 || keymatch == -1) {
   redisContext *c = connect_to_redis(server, port);
   redisReply *reply = redisCommand(c,"SET %s %s",key,value);
   reply = redisCommand(c,"EXPIRE %s %d",key,expiration);

   //Check the published values
   reply = redisCommand(c,"GET %s",key);
   if (reply->len == 0) {
      keymatch = -1;
      printf("Invalid Key.\n");
   } else {
      keymatch = 0;
      valuematch = strncmp(value,reply->str,DHTMSGLEN);
      freeReplyObject(reply);
      redisFree(c);
   }
   if (valuematch != 0 || keymatch == -1) {
      printf("Republishing key to redis.\n");
   }
}
```

Code Sample A.11: Querying the DHT analog for a message using *libhiredis*.

```
redisContext *c = connect_to_redis(server, port);
redisReply *reply = redisCommand(c,"GET %s",key);
if (reply->len == 0) {
   fprintf(stderr,"Requested a non-existent key.\n");
   return 1;
}
strcpy(value,reply->str);
freeReplyObject(reply);
redisFree(c);
```

A.3 Initialization Loop

Figure A.1 provides a call graph for the implementation's initialization loop. `mt6d_init_loop` is a *libev* loop that contains all of the other code, including other *libev* timers and watchers. The key elements of the loop are the `mt6d_init_conf`, which executes initialization tasks, `mt6d_config_rotate_cb`, which is the callback for the configuration rotation timers, and `mt6d_tun_pkt_cb`, which is the callback for the file descriptor watcher that is waiting to transform outbound packets.

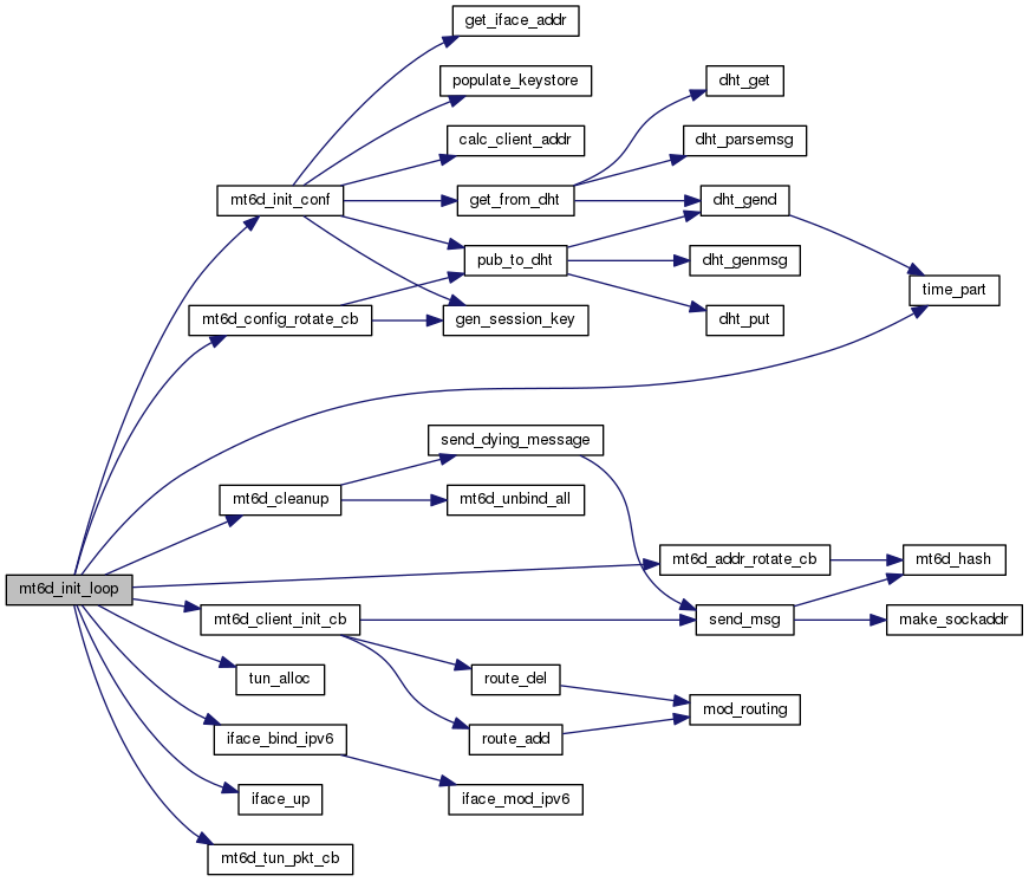

Figure A.1: The MT6D Initialization Loop Call Graph

A.4 Initialize Configuration

Figure A.2 shows the call graph for the configuration initialization function. This function provides all of the necessary configuration initialization functions, including publishing to or retrieving from the DHT, accessing the keystore, and generating the session key for the server.

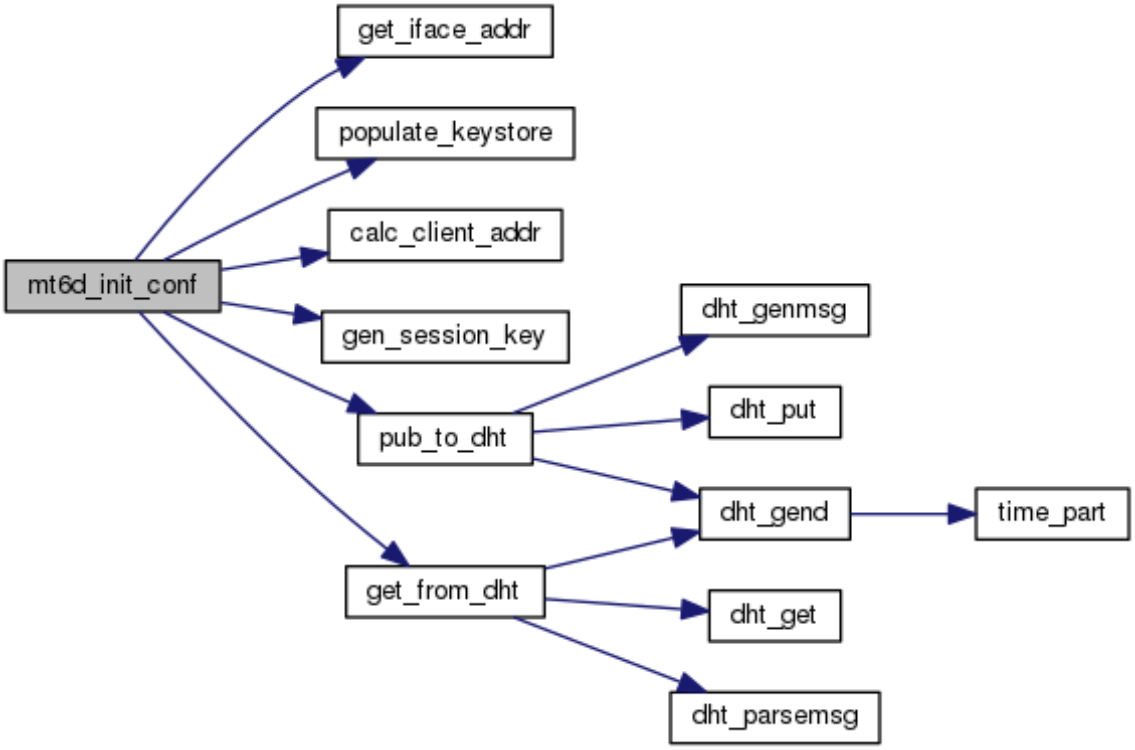

Figure A.2: The MT6D Initialize Configuration Call Graph

THIS PAGE INTENTIONALLY LEFT BLANK.