Implementation of network moving target defense in embedded systems

by

Robert Finstad

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Cyber Security

Iowa State University

Ames, Iowa

2020

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my committee chair, Douglas Jacobson and my committee members, Thomas Daniels and Ahmed Kamal for their guidance and support throughout the course of this research.

In addition, I would also like to thank my friends, colleagues, and the department faculty and staff for supporting me and my learning experience at Iowa State university. I appreciate the work of the university and those who are a part of it in providing an environment in which I was able to grow, both intellectually and in my field of study.

ABSTRACT

Moving target defense provides opportunities for adaptive defense in embedded systems. A great deal of work has been done on incorporating moving target defense techniques into enterprise systems to increase the cost to attackers and level the playing field. A smaller body of work focuses on implementing these techniques in embedded systems, which can greatly benefit from adaptive self-defense techniques. This work implements a network shuffling proof of concept in the Zephyr real time operating system to tackle the challenge of incorporating shuffling techniques into embedded systems. A host-centric, high security implementation is provided which maximizes attacker uncertainty and minimizes the impact of host compromise. Identifiers are utilized at the datalink, network, and transport layers and rotated per connection using keys shared between host pairs.

Existing shuffling schemes are explored, including those targeted to IoT contexts. Existing limitations in protecting embedded systems are considered along with the presented by moving target defense. The design details and implementation of incorporating a moving target defense module to in the Zephyr networking stack is provided. The protection provided by the scheme is evaluated and it is compared to existing address shuffling schemes. Future work in better handling data forwarding and collisions in the proof of concept scheme are considered. Options for adapting and building on the scheme to meet the needs of system designers are explored. This work provides system designers with insights into implementing address shuffling in embedded systems.

CHAPTER 1. INTRODUCTION

Embedded systems are becoming increasingly connected in a world in which adversaries have the advantage of carefully planning and carrying out attacks while defenders scramble to plug holes. Embedded systems are particularly difficult to protect as they do not have the same tools and techniques readily available in enterprise systems. Newer adaptive protection techniques being developed in traditional computing contexts provide opportunities for embedded systems to actively resist attacks. Network address shuffling, a popular moving target defense technique, allows embedded systems to significantly increase the cost of attacks. Implementing moving target defense techniques poses additional challenges in embedded systems but allows for adaptive defense to be employed.

Significantly less work in moving target defense has focused on embedded systems compared to enterprise system. This paper explores challenges and opportunities in implementing network shuffling in embedded systems, using the open-source Zephyr real time operating system. Embedded systems in high security contexts such as sensitive components in defense platforms warrant the protection provided by moving target defense. Due to hardware limitations in embedded systems, existing host software is modified to implement the scheme. A great deal of work in moving target defense remains conceptual whereas this paper aims to consider implementation from a system designers' perspective. Working through integrating shuffling techniques in the Zephyr network stack provides insight into how these techniques can be integrated in embedded systems. The implemented proof of concept reveals practical issues in employing moving target defense in embedded contexts.

Chapter 2 of this paper provides an overview of network moving target defense including existing shuffling schemes. Chapter 3 focuses on challenges and opportunities of implementing shuffling in embedded systems. Chapter 4 provides background on Zephyr and covers the implementation of shuffling in the Zephyr networking stack with a moving target defense module. Chapter 5 evaluates the implemented proof of concept including the protection it provides, options for adapting the scheme to different system designer’s needs, future work to resolve issues with wider deployment of the scheme, and a comparison of the scheme to existing shuffling schemes. Chapter 6 summarizes and concludes this work.

CHAPTER 2. NETWORK MOVING TARGET DEFENSE

2.1 Overview of Network Moving Target Defense

Moving Target Defense seeks to level the playing field between attackers and defenders by creating adaptive defenses that increase the cost of attacking a system. Traditionally, attackers have a great deal of time and resources to analyze static targets to find vulnerabilities and carry out attacks. Defenders must continually discover and mitigate vulnerabilities throughout entire systems and attempt to detect attacks by adversaries. Adversaries need only find a few holes across many interconnected systems to meet their goals, while defenders struggle to find and mitigate all issues in their system and detect attacks. Moving Target Defense (MTD) disrupts this paradigm by making systems dynamic rather than static. Continuous changes to attack surface remove the static view of systems and vulnerabilities normally gathered by adversaries prior to attacks. Systems can also change dynamically during attacks, making it difficult for them to succeed.

Network Moving Target Defense (NMTD) primarily focuses on shuffling network identifiers to make it difficult for adversaries to establish a consistent view of the network and target hosts. Identifiers such as ports and addresses are mutated periodically to make it appear as if the hosts on the network and their interactions have changed. Systems that attempt to detect attacks may also mutate in response to a potential attack to further confound adversaries. Many schemes make use of a central controller that either handles translation transparently via the network infrastructure or via both network infrastructure and host agent interaction. NMTD Schemes generally rely on the use of software defined networking and or custom host agent software to handle translation and mutation. Frequency of mutation, network protocols used, and identifiers that are mutated vary based on the scheme and the implementers preferences. More frequent rotation increases overhead but limits how long adversaries have current information on the network. Using IPv6 instead IPv4 allows for much larger address spaces in which to perform mutation and is a common proponent of schemes. Adding ports to the MTD scheme in addition to addresses increases the identifier space and creates additional uncertainty about network services running on hosts.

Comprehensive reviews of current research in moving target defense are provided by [1] and [21]. A review focusing specifically on network moving target defense is provided in [20]. A variety of existing moving target defense schemes are evaluated in [19]. Key properties required for an effective MTD scheme are explored in [11]. The vulnerability of network moving target defense to host profiling and traffic analysis are shown in [10], along with suggestions to mitigate the effectiveness of these analysis techniques. A variety of NMTD schemes are reviewed in the following section to present an overview of existing implementations.

2.2 Network Moving Target Defense Schemes

MT6D [5] is a distributed network moving target defense scheme that makes use of shared symmetric keys to derive IPv6 addresses. MT6D encapsulates the original packets in a tunnel and allows for the use of gateways or host only implementation. Improvements to MT6D are explored in [2], making use of a modified implementation of mobile IPv6 to resolve issues with address collisions and connection synchronization in MT6D. RPAH [3] implements a key pair and gateway coordinated scheme in which server host agents handle port translation while network gateways transparently handle network address translation as well as port translation for clients. Only currently valid address combinations between hosts that share keys are allowed through gateways. SDMA [13] performs routing based on address tokens in which valid address tokens are provided to authorized hosts via DNS server. Host agents handle translation of ports and network addresses.

PHEAR [12] makes use of packet header randomization to make connections anonymous to unauthorized network users. Addresses are random nonces and communications are managed by gateways and an SDN controller. ORHM [8] makes use of OpenFlow and SDN controllers to transparently provide rotating virtual IP addresses for hosts. Forwarding devices automatically handle the address translation on behalf of hosts. Senders receive the current virtual IP address for a remote host from a DNS server that interacts with the SDN controller. An SDN scheme that performs IP and MAC address rotation on end hosts via the use of NAT rules and host OpenFlow agents is provided in [9]. The SDN controller coordinates addresses and address rotation across network elements. RHSM [6] makes use of an SDN controller to coordinate network address and port mutation. FVRM [4][7] implements a similar scheme that also performs network address and port mutation via an SDN controller but provides unique IP addresses per each connection on a host rather than a single IP per host.

Micro MT6D [14][16][18] explores implementing MT6D in IoT devices including adapting it to 6LoWPAN and using lightweight cryptographic algorithms. An MTD scheme in wireless sensor networks is proposed [15] that makes use of single shared key on the network which is used in an HMAC function along with node identifiers and period data to derive addresses. This scheme incorporates a central controller to precompute addresses for each round and provide them to each node while also resolving any collisions. An MTD scheme for mobile ad-hoc networks is proposed in [17] that generates addresses using a hash chain based on a network-wide shared secret and node identifiers. Nodes choose when to change their address and broadcast their identifier and hash index to allow other nodes to calculate their address. Multi-hop routing and dynamically joining networks is also supported.

CHAPTER 3. SHUFFLING DEFENSE FOR EMBEDDED SYSTEM PLATFORMS

3.1 Embedded Systems

Embedded systems use special purpose computers designed to operate as part of a larger device or platform. They fulfil a variety of purposes and are widely used in devices requiring automated functions including vehicles, medical equipment, industrial equipment, and telecommunications equipment. The size, weight, and power consumption of these devices are limited based on the larger system they are a part. Embedded systems are commonly designed to control physical equipment and interact with the environment, requiring real-time capabilities and specialized hardware and firmware to accomplish tasks. As a result, embedded systems primarily implement low-level software and provide limited support for software libraries, frameworks, and programming languages. There are greater constraints on introducing additional hardware in embedded platforms and limitations to the amount of software compared to general purpose systems used in enterprise environments. Security solutions have few implementations in embedded systems compared to their wide availability in enterprise solutions.

The combination of hardware and software limitations in embedded systems and limitations of adding additional components make it difficult to use existing security solutions. Existing solutions and implementations need to be redesigned and adapted to special purpose systems with specialized requirements. As embedded systems become increasingly connected, many of the traditional problems facing networked systems threaten these devices and platforms, requiring existing concepts and schemes to be adapted and implemented to protect these devices in their current environments. Although the nature of embedded systems generally requires lightweight security techniques, embedded systems in the most sensitive contexts such as sensitive modules in defense systems require much greater protection.

3.2 Potential of NMTD in Embedded Systems

Embedded systems have limited security protection mechanisms compared to traditional enterprise systems but are commonly deployed in sensitive contexts, such as military systems. Non-traditional deployments have limitations to the amount of hardware that can be included and how it can be arranged within the system. Focusing on integrating network moving target defense shuffling techniques into existing end devices minimizes the need to introduce additional hardware. Additionally, developing shuffling techniques for the underlying system allows deployment in current operational contexts without requiring complex new technologies that may have limited implementation, such as software defined networking.

Real time operating systems are commonly used to control hardware in complex systems and are becoming increasingly connected as embedded systems utilize networks to share data and allow remote control. Network protection capabilities in these systems are limited, such as only providing Transport Layer Security to protect data and commands in transit. Host systems rarely provide state of the art protection against network attacks. Standard network protection solutions such as host-based firewalls and intrusion detection and prevention are rarely implemented in real time operating systems.

Network moving target defense shuffling techniques have potential in protecting embedded devices from network threats by preventing unauthorized hosts from being able to communicate with the device. To communicate with the embedded device and launch network attacks against it, an attacker must be able to locate and address the device. Implementation of shuffling techniques increases the difficulty of locating network devices. When devices are located, they only appear at the discovered address for a limited period, requiring an attacker to rediscover them after each mutation. Depending on the shuffling scheme and its implementation, an attacker's ability to target specific devices can be severely limited.

3.3 High Security NMTD Scheme for Embedded RTOS

The network moving target defense scheme proposed in this paper is intended for deployments with stringent security requirements in the protection of networked embedded systems. Significant overhead is introduced to actively cycle through network addresses with each host involved in communication to provide minimal opportunities for attackers. To provide sufficient address space and maximize uncertainty about the identity of network devices, data-link addresses and transport layer ports are included in the shuffling in addition to network addresses. This creates limitations in forwarding depending on the underlying medium and technologies deployed. The current implementation is best suited to networks in which devices can directly send data-link frames to one another (e.g. hub networks, bus networks, and mesh networks). Future work and adaptations to better support forwarding are considered in section 5.

Communicating hosts share a cryptographically random key that is used along with connection and period specific information as input to an HMAC algorithm. The output of is used to derive the addresses used by each side of the connection for the current period. The MTD scheme provides host-based network access control as it will only accept frames that match the expected incoming datalink and network addresses and transport layer ports and protocol type expected. The default implementation only allows traffic on MTD connections, so any outgoing or incoming frames that do not match any MTD connection are automatically dropped, preventing unauthorized communications. Devices are expected to be preconfigured with shared keys prior to deployment based on the connections required by system applications.

CHAPTER 4. IMPLEMENTING SHUFFLING IN ZEPHYR

4.1 Overview of Zephyr

The Zephyr Project [24] is a Linux Foundation Project to provide an open source community governed RTOS to meet the needs of embedded system and IoT developers. Zephyr presents itself as an alternative to commercial RTOS offerings while allowing vendors to be involved in shaping its development to meet the needs of embedded systems and IoT communities. Zephyr provides a lightweight RTOS kernel as well as a collection of drivers and subsystem modules to support a wide variety of applications with a limited footprint.

The Zephyr Project provides a long-term support (LTS) branch to provide a stable API and ongoing support for an extended period. Extended security updates, maintenance and support on a stable version provides a useful baseline for system integrators. Due to the benefits of LTS, the Zephyr LTS branch 1.14 has been selected for implementing NMTD shuffling techniques. The 1.14 branch includes minor versions for the inclusion of fixes and patches in the form 1.14.x. 1.14.2 is the specific version used during development as it was the latest available at the time of development. Documentation on developing applications with Zephyr as well as its components and APIs are provided under the documents portion of its website for the selected version [23]. Note that version 1.14.1 of the documentation is used for the 1.14.2 branch at the time of writing.

4.2 Zephyr Networking Stack

Zephyr provides support for commonly used network protocols and technologies including the TCP/IP networking stack, TLS, Ethernet, Bluetooth, Wi-Fi and 802.15.4/6LoWPAN. Additional application protocols commonly used in IoT are also included such as MQTT and CoAP. For the initial implementation of the scheme, Ethernet as used as the underlying data-link protocol for transferring data frames because it can be easily used in emulated environments. The QEMU environment for developing and testing Zephyr applications provides support for Ethernet communication using Linux virtual networking capabilities. Zephyr also provides the standard Berkeley sockets API to facilitate development of network applications.

Unlike Linux, Zephyr does not provide network hooks and user space application support for features such as packet filtering and translation. To provide desired network behaviors in Zephyr that are not part of current network stack implementation, additional code must be introduced into the network stack. The Zephyr networking stack implementation is loosely organized as sockets, grouped kernel protocol modules for higher level networking protocols (e.g. application protocols, TCP, UDP, IPv4, and IPv6), a network interface module, grouped link-layer protocol modules (e.g. Ethernet), and network device drivers. There is also a core networking module that determines the appropriate module calls to make for processing data flowing through the network stack (e.g. data requested to be sent by sockets, or data waiting to be processed after being provided by a network device driver).

Figure 1 [22] provides a high-level view of how incoming data flows through the network stack using the example of an incoming UDP packet. The Ethernet driver receives the data and places it in a queue to be processed by the kernel. The kernel takes in the packet in its core module, determines it is an Ethernet packet, then performs ethernet connection matching and header processing. The packet is then processed through the appropriate network interface, sending it through the appropriate network and transport protocol handling modules based on the packet’s headers while also matching the identifiers to a known connection. Once this processing is complete the packet data is queue in the appropriate socket for the application to read. Each module strips or iterates past headers as appropriate, using buffer and packet memory handling functions.

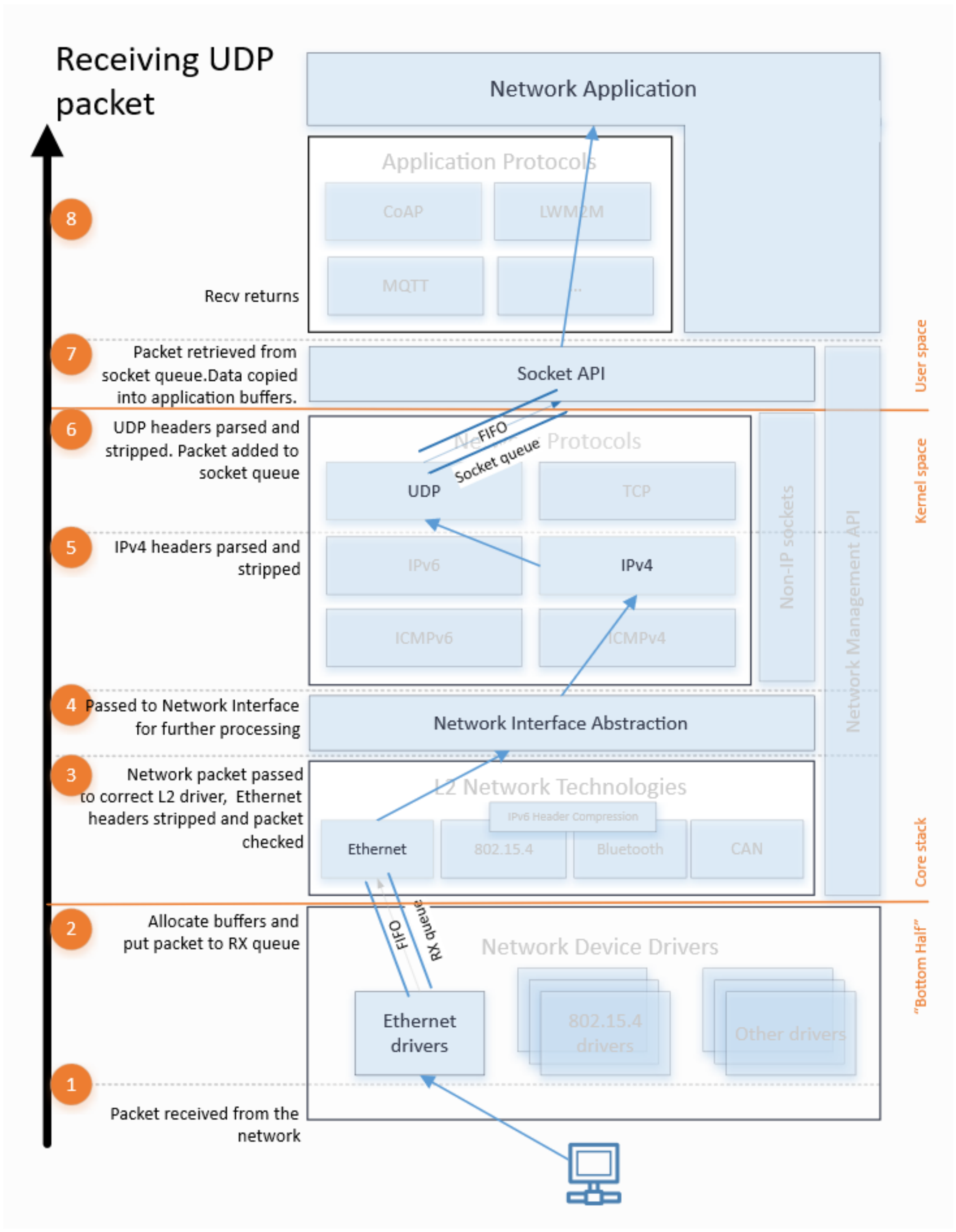
Receiving UDP packet
Network Application
Application Protocols
CoAP
LWM2M
MQTT
..
8
Recv returns
7
Packet retrieved from socket queue.Data copied into application buffers.
Socket API
6
UDP headers parsed and stripped. Packet added to socket queue
FIFO
Socket queue
Network Protocols
UDP
TCP
5
IPv4 headers parsed and stripped
IPv6
IPv4
ICMPv6
ICMPv4
Non-IP sockets
Network Management API
4
Passed to Network Interface for further processing
Network Interface Abstraction
3
Network packet passed to correct L2 driver, Ethernet headers stripped and packet checked
L2 Network Technologies
IPv6 Header Compression
Ethernet
802.15.4
Bluetooth
CAN
2
Allocate buffers and put packet to RX queue
FIFO
RX queue
Network Device Drivers
Ethernet drivers
802.15.4 drivers
Other drivers
1
Packet received from the network
User space
Kernel space
Core stack
"Bottom Half"

Figure 1. Zephyr Network Stack Receive Flow

A similar process occurs in reverse when a socket writes a packet to be sent over the network, as shown in figure 2 [22]. Based on the type of socket and the interface it is bound to, the appropriate transport and network protocol modules are used to prepend protocol headers to the data. The packet is provided to the network interface which ensures the interface is in the correct state and calls the appropriate data-link handling module to prepare the frame for the network device driver. The network device driver handles sending the frame on the network.

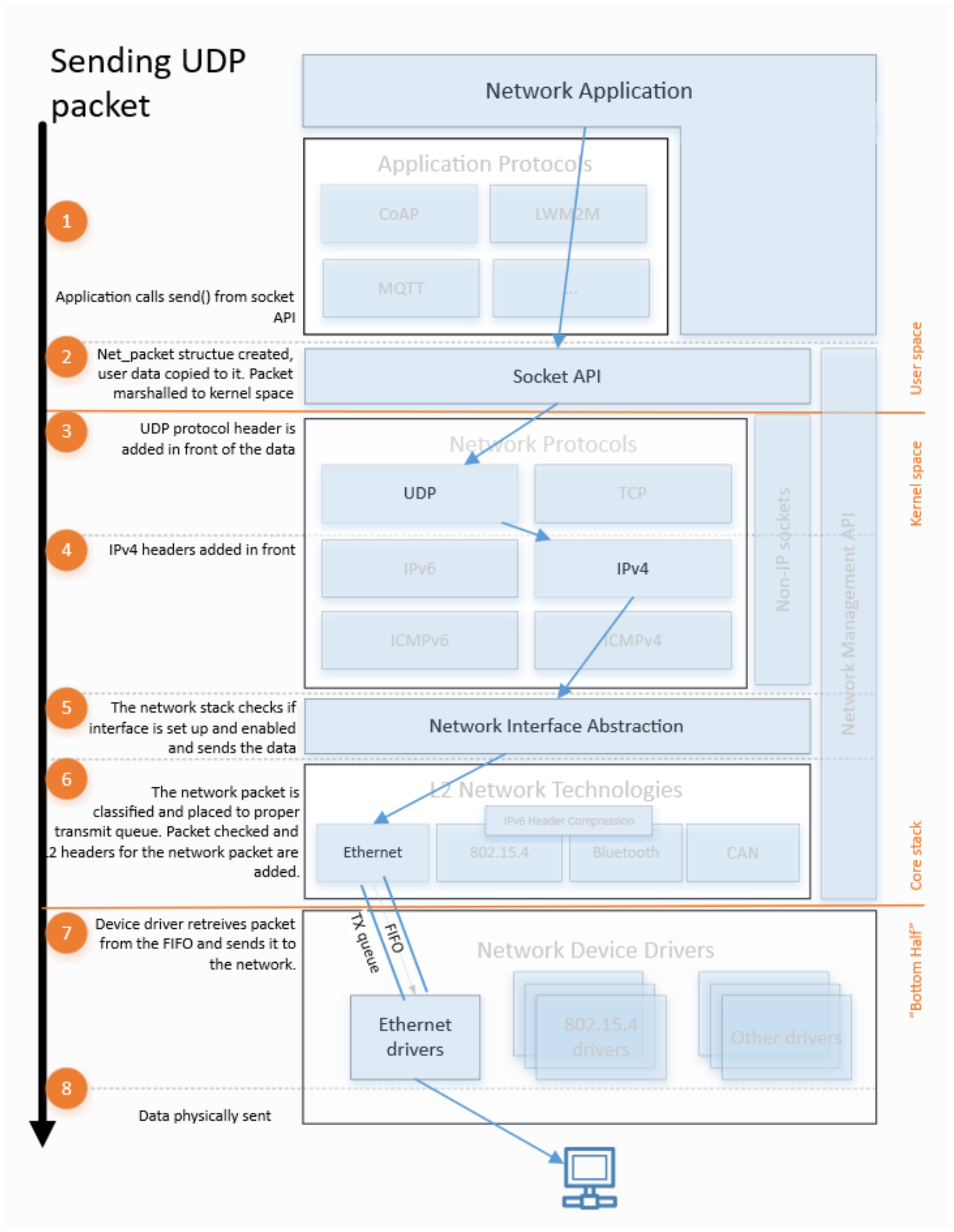
Sending UDP packet
Network Application
Application Protocols
CoAP
LWM2M
MQTT
...
1
Application calls send() from socket API
2
Net_packet structue created, user data copied to it. Packet marshalled to kernel space
Socket API
User space
3
UDP protocol header is added in front of the data
Network Protocols
UDP
TCP
4
IPv4 headers added in front
IPv6
IPv4
ICMPv6
ICMPv4
Non-IP sockets
Network Management API
Kernel space
5
The network stack checks if interface is set up and enabled and sends the data
Network Interface Abstraction
6
The network packet is classified and placed to proper transmit queue. Packet checked and L2 headers for the network packet are added.
L2 Network Technologies
IPv6 Header Compression
Ethernet
802.15.4
Bluetooth
CAN
Core stack
7
Device driver retreives packet from the FIFO and sends it to the network.
TX queue
FIFO
Network Device Drivers
Ethernet drivers
802.15.4 drivers
Other drivers
"Bottom Half"
8
Data physically sent

Figure 2. Zephyr Network Stack Send Flow

Of note is the split between upper level network protocol processing and data-link network protocol processing. There is a stopping point at the network interface abstraction where the packet is queued to be handled in another work thread in the upper network protocol modules (for incoming packets) or data-link network protocol modules (for outgoing packets). Similar splits occur between the socket and upper layer protocol modules and between the data-link protocol modules and network device driver. Within each module group, processing is otherwise sequential and follows through all modules within the group to complete requested network processing.

4.3 Zephyr Emulation with QEMU

The Zephyr build framework has built in support for running system, images in a QEMU virtual machine. Both Zephyr and QEMU support x86 processors with the Intel e1000 ethernet driver. QEMU provides an emulation of an e1000 ethernet NIC while the underlying Linux OS supports Ethernet traffic over virtual network interfaces using the Linux TAP driver. To support multiple Zephyr VMs communicating with one another and the Linux host machine, the example virtual network interface setup script provided with the Zephyr networking tools was modified to create a TAP interface for each Zephyr VM. The script was also modified to create a bridge interface to connect each TAP interface together, placing all Zephyr VMs on a shared virtual network also available to the Linux host. Each VM is configured to attach to its own TAP interface when running.

4.4 Promiscuous Mode Support

To facilitate implementation and testing, the system is run in QEMU emulating an x86 board. This allows the use of the Intel e1000 driver supported by both QEMU and zephyr for Ethernet connectivity. Further investigation into the promiscuous mode capability of Zephyr revealed that the networking subsystem had support for this in the kernel but that most Ethernet drivers in zephyr did not support enabling promiscuous mode, including the e1000 driver. Unicast promiscuous mode is required for the proposed network moving target defense scheme as a single ethernet interface needs to be able to receive frames for multiple ethernet addresses including those that are not registered on it. This is also the case because Zephyr only supports setting a single MAC address per Ethernet interface currently.

Although unicast promiscuous mode was not implemented in the e1000 driver in Zephyr, the e1000 ethernet adapter itself supports it. To support the proposed NMTD scheme, the e1000 driver in Zephyr was modified to support enabling unicast promiscuous mode (UPE). Doing so required the addition of code to set the correct values in the underlying register to enable UPE as well as registering the newly added promiscuous mode capability and Ethernet driver API calls to the Zephyr networking subsystem when initiating the e1000 driver. With this addition, the driver now supports providing all unicast ethernet frames to the networking subsystem for further processing in the NMTD implementation.

To simplify implementation and because the NTMD scheme always requires promiscuous mode to be enabled, the driver automatically enables it and leaves it on while active. The promiscuous mode API requires that the driver support the set config API for the ethernet promiscuous mode command, so the current implementation returns success to calls to enable promiscuous mode to allow the rest of the network stack to enable promiscuous mode. This entire feature is conditionally compiled based on the selection of promiscuous mode support when building Zephyr. Similar updates will need to be made to other Zephyr Ethernet drivers to make use of this NMTD scheme on boards using different Ethernet adapters if not already supported.

4.5 Network Moving Target Defense Module

Applications establish a socket that behaves as if it is sending to and receiving from a consistent port and address. When sending packets, the MTD logic maps the outgoing port/addresses to the current random port/addresses for the end host and sends the packet out to the network. This involves updating TCP/UDP headers and IP headers, including checksums. When a frame comes in, it is passed to the MTD module in the network stack prior to normal network processing. The datalink, network, and transport layer headers are parsed by the MTD module and compared against known MTD connections. If a match is found, the Ethernet, IPv4 and TCP or UDP headers are translated to those expected by the end application and checksums are updated as needed before sending it up the normal network processing stack.

When data is sent to a remote host, it is processed through the network stack normally and makes a call to the MTD module before being sent out to the Ethernet driver to be sent on the network. The MTD module compares the sending network context against registered MTD network contexts. If a match is found, the packet’s headers and checksums are updated before the buffered frame is passed to the Ethernet Driver. ARP is bypassed when sending using an MTD connection. the sending handler determines and sets the destination Ethernet address based on the MTD translation table.

The scheme also provides access control based on registered MTD connections with another host. If no match is found for an incoming frame or outgoing network context, the function call to the MTD module returns an error and the network stack drops the frame. To ensure only authorized packets are sent and received, all relevant packet headers are checked together and must match all expected identifiers. Due to the complexity of the existing network stack and multiple calls to the same functions for a variety of purposes, a single entry/exit function call is made within the stack to perform checking and translation. This also provides the benefit of more easily updating the MTD module in the future since most processing is handled in one place as opposed to needing to update many hooks as the network stack implementation of Zephyr continues to be updated.

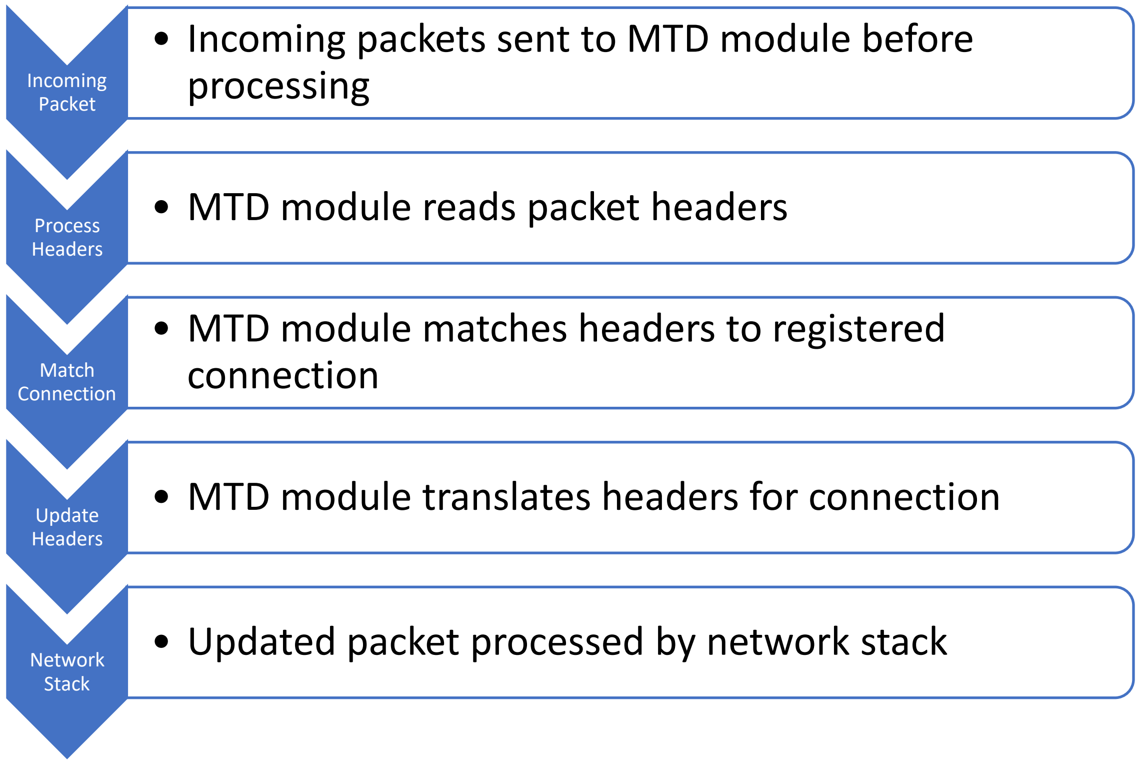
Incoming Packet
Incoming packets sent to MTD module before processing
Process Headers
MTD module reads packet headers
Match Connection
MTD module matches headers to registered connection
Update Headers
MTD module translates headers for connection
Network Stack
Updated packet processed by network stack

Figure 3. MTD Module Incoming Packet Processing

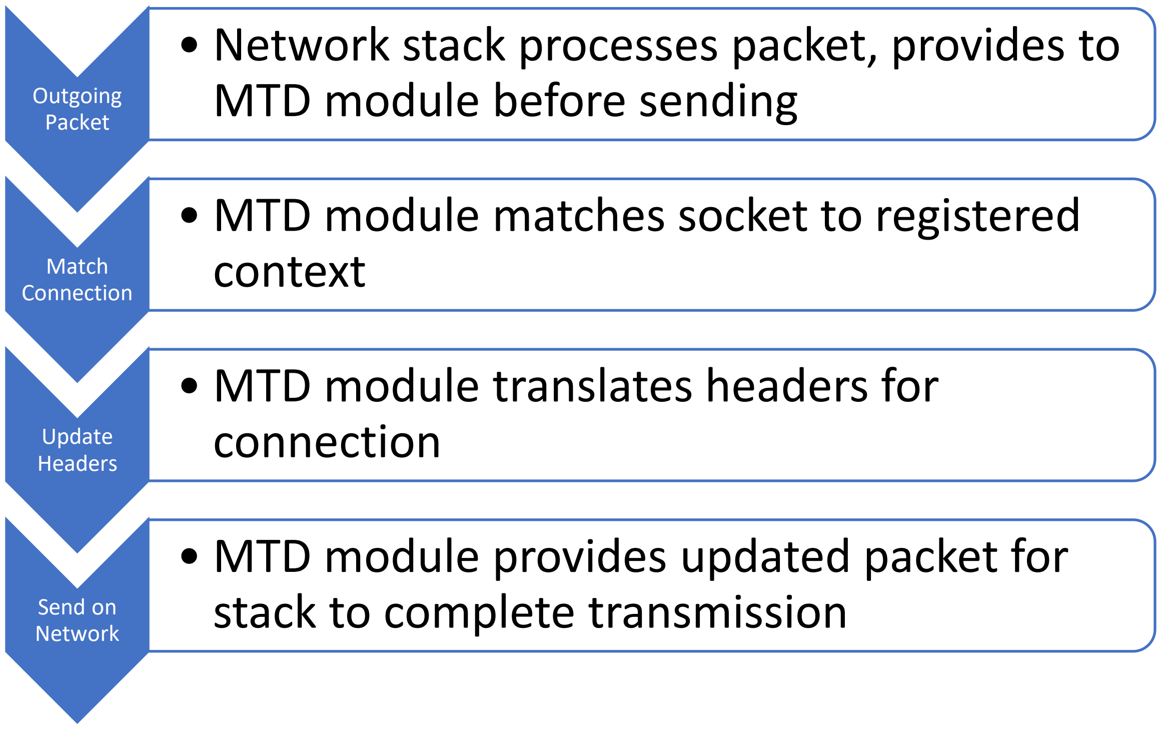
Outgoing Packet
Network stack processes packet, provides to MTD module before sending
Match Connection
MTD module matches socket to registered context
Update Headers
MTD module translates headers for connection
Send on Network
MTD module provides updated packet for stack to complete transmission

Figure 4. MTD Module Outgoing Packet Processing

Whenever the network stack dynamically registers or unregisters a connection it calls the corresponding function in the MTD module which handles matching the connection to a registered socket that is implementing MTD. The network stack removes and re-adds specific connections over time based on protocol state (i.e. TCP connection state) and the current status of a socket, which is mirrored by calling the MTD module for every register and unregister to ensure it has a consistent state with the rest of the network stack. The socket's network context is kept in the translation tables as a stable MTD registration. The network context associated with the socket also provides information on the existing connections for the socket, which are used during the MTD connection registration. The translation tables are stored per socket, then applied to the connection based on matching it to a registered MTD socket network context when a connection registration call occurs.

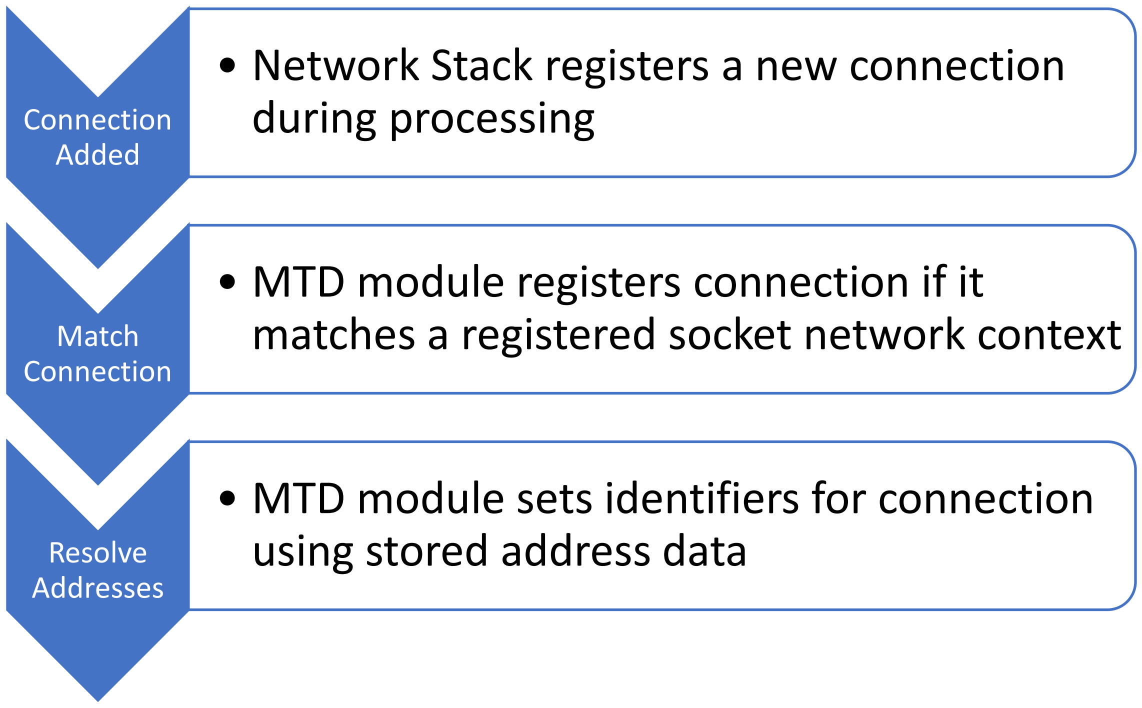
Connection Added
• Network Stack registers a new connection during processing
Match Connection
• MTD module registers connection if it matches a registered socket network context
Resolve Addresses
• MTD module sets identifiers for connection using stored address data

Figure 5. MTD Module Connection Registration Hook

To setup an MTD socket, the user application makes an MTD register system call in which they provide a registered socket file descriptor, the shared key to be used for the connection, local context data, and remote context data. The local and remote context data are used along with the key to derive period specific addresses for the local and remote host, respectively. The caller is responsible for ensuring the local and remote context data are not the same. The application is granted control and flexibility in the scheme it chooses to implement for the context data. The application may for example use a keyword such as a host name or descriptive string combined with a counter or a timestamp. This allows the application developer to choose the scheme for rotating addresses as some implementations may not support typical schemes (i.e. reliable time may not be available in some embedded systems). The developer provides application context-dependent mutation data. Reference schemes are discussed in section 5.

An MTD address rotation system call is provided which takes the registered socket file descriptor and updated local and remote context data. Both upon initial registration and upon calling the address rotate system call, the provided key is used in an HMAC function over the local and remote context data to derive the local and remote addresses, respectively. The MTD module makes use of the MbedTLS library provided with Zephyr to provide implementation of cryptographic functions. This is like the TLS socket option feature that is implemented in the Zephyr kernel includes MbedTLS to allow applications to make use of TLS more easily. The output of the HMAC function is used to set random Ethernet and IP addresses as well as transport layer ports. Due to a limitation in the Linux kernel, the unicast address bit is masked to always be zero. The Linux kernel does not support treating ethernet frames with a multicast address bit set as unicast frames and will attempt to automatically perform multicast registration and forwarding, causing issues with connections. Further testing is required on real hardware to determine if this bit can be random in practice.

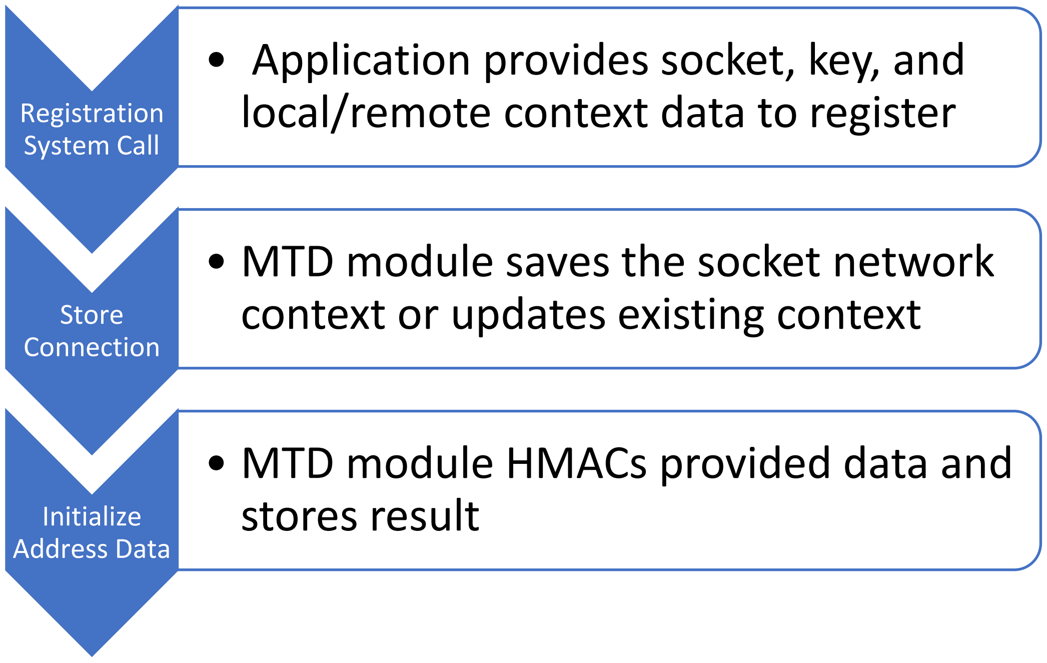
Registration System Call
• Application provides socket, key, and local/remote context data to register
Store Connection
• MTD module saves the socket network context or updates existing context
Initialize Address Data
• MTD module HMACs provided data and stores result

Figure 6. MTD Module Registration System Call

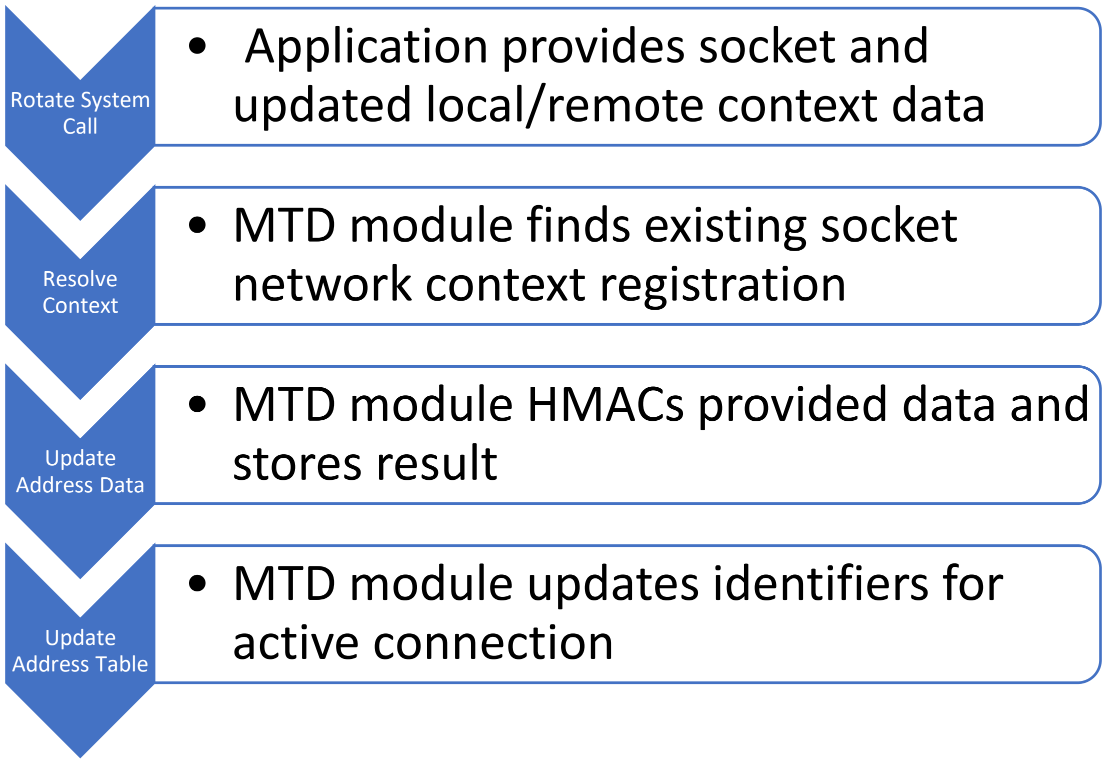
Rotate System Call
• Application provides socket and updated local/remote context data
Resolve Context
• MTD module finds existing socket network context registration
Update Address Data
• MTD module HMACs provided data and stores result
Update Address Table
• MTD module updates identifiers for active connection

Figure 7. MTD Module Rotate System Call

TCP posed a greater challenge than UDP in handling socket registration. When a TCP socket listens, it automatically spins off a new socket during the TCP handshake and will begin sending using the new socket before the socket is returned to the application. To handle this case, a hook has been added to the TCP syn-ack handler to call an MTD function that manages this transition. The function is provided with the previous sockets context and the new socket context that is being registered, and copies the key and context data from the original socket into the newly created TCP socket being used to send and receive data. The socket connection to the client can be used immediately with the same connection details in the MTD scheme. Applications must use a single listening socket per MTD TCP connection as the scheme is not designed to serve multiple clients with the same initial connection. It is recommended that applications close the original listening socket after the client establishes a connection and reopen it with the latest rotation context parameters after the client connection is terminated to allow a new TCP connection with the current MTD state.

Zephyr features are selected and included using Kconfig. To support MTD, additional entries were added to the kconfig options in the internet protocol kconfig options (the primary kconfig file for higher level network protocols) to choose to enable MTD. The MTD option entry has dependencies on promiscuous mode, MbedTLS, Ethernet, IPv4, UDP, and TCP. If any dependencies are missing, configuration warnings are presented and MTD is not compiled for the system image. The MTD module was also added to the network subsystem cmake source listing to conditionally include the MTD module source when compiling the kernel with MTD enabled. The network logging kconfig was also updated to add logging support for the MTD module, allowing the MTD module to log debug messages to assist in development and debugging.

4.6 NMTD Test Applications

To test the MTD implementation, two example Zephyr applications were created, an echo client using MTD, and an echo server using MTD. The applications are created in the style of Zephyr sample programs, allowing them to be compiled like other sample socket applications under Zephyr using the Zephyr build framework. The projects are configured to build the core networking functionality of Zephyr including Ethernet, IPv4, UDP, and TCP. The configuration also enables MbedTLS, MTD, promiscuous mode, the e1000 ethernet driver, and network stack debug logging. A macro is used to define if UDP is being used instead of TCP when compiling the example applications.

The server application sets up a listening socket and waits for the client to connect and send data. Upon receiving data, the server sends the data back to the client. The client program sends data patterns of increasing length to the server and closes the connection after completing the sequence. In each application, after creating the initial socket but before binding it, the mtd_register system call is invoked with the socket file descriptor, a shared test key, and a context data string of “server” for the server portion of the context and “client” for the client portion of the context. The client and server bind their sockets and connect to one another using arbitrary addresses in the application and proceed to send and receive data. After each send and receive, both applications call mtd_rotate, providing the socket file descriptor, and the original context strings with a counter appended to them, which is incremented for each call.

The actual address that the sockets bind and connect to does not matter in the application as they are not used to communicate on the network. Once the registration takes place and data begins being sent on the network, the MTD module automatically translates the outgoing headers to have the appropriate identifiers. As the applications run, they output debug log messages showing the results of registering MTD connections and the translation of incoming and outgoing packets between the original identifiers the socket expects to see and the identifiers derived using the shared key and context data in the MTD module. Each application indicates when it sends and receives a message.

CHAPTER 5. EVALUATION AND FUTURE DIRECTION

5.1 Protection Provided by Approach

The MTD scheme implemented provides general protection against network attacks by impeding network reconnaissance and the ability to target specific hosts. By only accepting expected addresses during a rotation period, the scheme provides access control that prevents adversaries from reaching the host entirely. Adversaries perform network reconnaissance primarily by scanning hosts on a network and detecting what services are running on each host to potentially exploit. An adversary will be severely limited in attempting to scan any host that implements the moving target defense scheme. To successfully discover a single service, the adversary needs to use the correct set of identifies made up of Ethernet and IPv4 address, and port, for both the source and destination portion of the connection. This makes up two 48-bit ethernet addresses, two 32-bit IPv4 addresses, and two 16-bit port numbers, providing entropy of up to 192 bits. Even if a service is successfully discovered using this method (i.e. the host is on the local network and able to take on any address it wants) the information will only be valid during the rotation period, after which it is necessary to rediscover the host.

The MTD scheme forces adversaries to adopt a more expensive strategy in which they must have access to monitor the network segment they are targeting and monitor ongoing traffic to detect valid source and destination address pairs. The attacker may be able to discern what service is being used during this period by analyzing the traffic, assuming it is unencrypted. The attacker may also attempt to impersonate one or both sides of the connection to determine the service being used, but this can again be mitigated using traffic encryption. If the attacker succeeds in identifying services used by the connection (i.e. encryption is not used), they may again attempt to impersonate the connection and launch an attack, but they are still limited to doing this within the same period when using traditional attacker techniques. Employing adaptive tools and techniques would allow the attacker to attempt to trace a connection across address changes and launch attacks that occur within a single period. They may also attempt to launch specialized attacks that can adapt to addresses changing in the middle of the attack. This forces the attacker out of the position of having long periods of time to analyze the network, gather information, and carry out attacks. Attackers must instead actively attempt to determine the state of the network and adapt to its changing state, greatly increasing the cost of attacks.

The most effective attacks require the adversary to compromise a node with a connection to another target. If the attacker can gain access to a host in the MTD scheme, they will be able to determine the interactions with other MTD hosts and may find vulnerabilities to exploit over the existing MTD connection. Another attack that remains viable, at additional cost, is performing network denial of service against a host. Regardless of the use of traffic encryption and or authentication, the attacker can monitor the network, find an active address pair, and send spoofed packets to force the remote hosts network stack to process the traffic. To maintain an attack against the same host, the attacker must employ advanced tools and techniques that attempt to determine the new address of the targeted host using traffic analysis and network profiling. This increases the cost of performing a denial of service attack by requiring tools that attempt to adapt the attack to the changing network addresses.

A significant weakness in the current implementation is that the network device driver is configured to provide all incoming traffic to the network stack for the network stack can determine if it is part of a valid MTD connection. Frames that would normally be dropped by hardware interface are now handled in software and dropped after initial processing if they do not match a known MTD connection. This implementation weakness comes from limitations in setting multiple addresses on network hardware. Network interface hardware that provides more advanced features and customization could be introduced along with a scheme update to manage the set of current link-layer addresses on the interface to reduce the impact of this additional processing. This hardware could be further adapted to offload portions of the MTD logic to hardware, like MT6D.

5.2 Handling Forwarding

The current implementation of the scheme is only intended to run in networks in which each host can directly send frames to another host without requiring an intermediate device to make forwarding decisions. Smart forwarding and routing of packets to hosts that are not directly reachable requires the scheme to be extended to provide wider network connectivity. Three approaches are proposed for extending the scheme to a wider network: introduction of one or more gateways, implementing the MTD scheme in forwarding devices, or limiting the address mutation scheme to be more compatible with existing infrastructure.

Many embedded systems are already designed to communicate through a central gateway to collect data in a single location, perform more expensive computations, and handle additional protocols and communication technologies not implemented in the embedded node. A gateway could also be a generic host device that implements the MTD scheme on at least one interface to communicate with MTD hosts and then handles collecting and forwarding any data and commands between the MTD nodes and other devices. The scheme could be more easily implemented on a standard OS, i.e., using the Linux netfilter framework and existing network tools (utilized by [9] and [12]) to match the scheme implemented in Zephyr. A Zephyr node or similar device could also be configured with a modified scheme to allow one or more interfaces to connect to another network while having MTD connections with each node in the network. Additional care must be taken in protecting the gateway in such an approach as it becomes a prime target for attackers and a method for reaching the rest of the MTD devices. Such a gateway could also implement a different MTD scheme on another network and translate between the two, acting as a proxy.

Another option is to implement the MTD scheme in the forwarding devices themselves. This approach would require a significant change to the normal behavior of switching and routing devices but could provide for forwarding random addresses in a controlled fashion. In such an approach, each forwarding device would implement the MTD scheme like a gateway described above and would share connections with downstream MTD devices. Devices would communicate with their immediate switch and router to register their current connection addresses. Routers would register MTD connections with one another to share routing information to perform MTD routing. Network devices become targets of interest in this scheme as they know information about connections between end hosts, but interaction with forwarding devices can be restricted by requiring a valid MTD connection. Such a scheme would be similar to implementing an OpenFlow agent on each host and forwarding device and having a controller handle coordination, but would be more decentralized with hosts and forwarding devices coordinating to forward data rather than a central controller making forwarding decisions.

A simple option that limits the effectiveness of the MTD scheme but allows it to be used with existing forwarding devices is to limit which layers perform address rotation and or what portions of an address are rotated. Disabling rotation of Ethernet addresses would allow normal switching to occur at the cost of significantly reducing entropy of the overall connection identifier and allowing adversaries target devices by MAC address (i.e. denial of service flooding). IP addresses could be limited to rotate within a subnet to ensure normal routing occurs, but this would also reduce the entropy of addresses depending on the size of the subnet and also allow adversaries to target subnets when scanning. This may be a desirable option to implement MTD when the security provided by additional address entropy is not warranted, overhead needs to be reduced, and forwarding capabilities are desired without introducing gateways or otherwise customizing network devices.

5.3 Combining with TLS

Using MTD alone does not protect the data contents of network connections. Aside from the data being unprotected from eavesdropping, the type of data being exchanged may be used to track which addresses belong to which connections during each period. An eavesdropper is also able to view the addresses and connection details being used for data and may be able to inject bad data into a connection by spoofing a valid source for that address mutation period. The address rotation reduces the period in which such an attack can occur, and in extreme cases can even prevent it (i.e. per message rotation) but is not suited to mitigating these issues without additional protection techniques, such as Transport Layer Security.

Zephyr has built in support for TLS connections which can be used to prevent unauthorized data from being injected into a connection and to protect the confidentially of data in the connection. In cases where implementing an MTD scheme providing access control is justified, or if additional protection on data content is warranted, TLS should be used by the network application to further protect communications. Spoofed data will no longer be accepted by the end application and eavesdropper will be unable to determine the contents of data to track connections over mutation periods. The combination of the MTD scheme and TLS provides significant mitigation against data disclosure, data tampering, and targeting of specific hosts. The combination of the implemented MTD scheme and TLS is not designed to prevent adversaries from identifying connections using advanced traffic analysis and profiling techniques but is expected to make them more costly.

5.4 Deciding When to Shuffle

Deciding when to shuffle is a key issue in implementing an effective MTD solution. More frequent address mutation reduces the window of opportunity for attackers but introduces additional overhead. Many of the solutions reviewed section 2 use time as the primary means to decide when to shuffle. This has desirable properties but poses challenges that have an even larger impact on embedded systems. Using time as a metric makes it easy to judge the window of opportunity for an adversary to act within a single period but requires that devices involved in the scheme have a reliable time source and remain synchronized. Although this is normally the case in enterprise systems, embedded systems may not keep track of real-world.

If the system can reliably obtain time on startup or is always able to keep a clock active to track time (i.e. the time component is always on in a low power state using a battery), a time-based approach to performing address rotation is still viable. There are two potential approaches depending on the nature of the system. If an external clock source is available and the system can synchronize to it, the entire system can maintain time synchronization to manage clock skew. This requires that the time source can be reliably reached either via MTD or another communication method. A possible alternative is to consider the rate of clock skew and build in additional tolerance. Allowing previous and next addresses to be maintained instead of just the current address can allow a wider window in which devices successfully send to one another at the cost of having a longer period in which each address remains valid. For long-lived systems, it would be possible to adjust time as part of regular system maintenance. The skew rate, system lifetime, and frequency and maintenance would allow tradeoffs decisions on the period to use.

A simple alternative is to base the rotation frequency on number of messages. After a certain number of messages are sent and or received on a connection the two systems could perform a rotation. This scheme is viable if the system can determine messages are being reliably delivered (e.g. counting acks), but in the event of a desynchronization, manual intervention may be required to allow the system to communicate again. If one system is counting messages the other never receives or trying to talk to an offline system and continuing to rotate its address, when the other system is finally available and can receive messages they will not be synchronized. An additional challenge when only counting reliable messages is that multiple messages may be sent, and an earlier message may cause a rotation that causes the remaining messages to fail. This could be handled by resending but is prohibitively expensive. In such a case, it may be desirable to accepting messages on the last rotations addresses for each period to mitigate this loss.

Some additional options to consider include making application context dependent decisions and implementing a shuffling control protocol. In a shuffling control protocol, the host that wants to rotate first could send a shuffle request message and provide a context data parameter for the peer to use. The peer can reply with a shuffle response message acknowledging the request and providing its own context data to use. To provide a more robust scheme, the two hosts could require the successful receipt of a shuffle complete message from each side (initiator, followed by peer upon receiving the initiators message) on the new address to verify successful rotation. If this verification fails, the hosts could fall back to the previous addresses and try again with different parameters. In an application dependent scheme, the developer chooses a desirable point in time to rotate addresses. The application protocol may for example choose to rotate at the completion of a transaction (e.g. completion of a CoAP request and response message pair or completion of sending an MQTT publish message and acknowledge pair, either of which may span a variable number of packets depending on data size).

5.5 Dealing with Collisions

Depending on the deployment scenario, additional work is needed to ensure the scheme does not result in connectivity loss due to address collisions. Assuming devices can directly send frames to one another, they should still be correctly accepted or rejected so long as at least one part of address (port, IP, MAC) is not also in collision. In this scenario, both hosts will receive the packet, but the host will only keep the packet if all source and destination address and ports match the expected MTD parameters for the connection. The probability of this occurring in the original scheme is extremely small as this would require two pairs of connections to resolve to the same 192 (191 excluding Ethernet multicast bit) bits worth of identifiers on the connection during a rotation period. The probability of collision increases if the allowed addresses are limited or some layers do not implement MTD. The probability of collision also increases based on the number of local MTD connections. Coexistence with non MTD nodes would also slightly increase the probability of collision and cause unwanted packets to be processed by non MTD interfaces when collisions occur. If a full identifier space collision occurs, packets will be incorrectly sent to and accepted by hosts not part of the active connection. The application layer may be able to mitigate this (i.e. TLS will still reject them for using the wrong authentication key) but this occurrence causes a much larger problem if forwarding also needs to be handled.

One solution for networks that need general forwarding capabilities is to build off the address change protocol considered in section 5.4. When a collision occurs that results in loss of connectivity, the hosts will fall back to the previous address and attempt to establish new addresses with different context data. The remaining problem is to deal with conflicting mutations that happen independently. For general purpose forwarding, the forwarding devices would need to understand and support the MTD scheme as discussed in section 5.2. An addition would be needed to both schemes in which upstream switches and routers detect an MTD address collision when hosts perform rotation (i.e. detect addresses already present in switching or routing tables coming from an incorrect port). While forwarding the handshake, a forwarding device which detects a conflict drops the handshake packets, causing the handshake to fail and the end hosts to attempt a handshake with new addresses. If multiple devices create the same conflict, it is expected that all of them would fail the handshake based on the offending addresses being cached as in use.

In the case where a central gateway is used for communication the gateway would be able to detect and address collisions because it maintains addresses with other MTD hosts on the network. To minimize work on non-gateway hosts the gateway can be responsible for initiating address rotations and providing full context data with a variation of the protocol proposed in 5.2. The gateway would be able to detect an address collision in advance when initiating an address mutation and if the parameters selected would cause a conflict it discards the initial context data and generates new context data until a conflict no longer occurs. This scheme ensures the gateway will be able to reliably tell the difference between two connections that end up colliding.

5.6 Options for Lower Security Systems

The existing MTD implementation can be adapted for systems that have lower security requirements, allowing greater interoperability and or reducing overhead. If desired, non MTD connections could be allowed on MTD hosts. A particular link could be marked as trusted and whitelisted with a known static set of addresses to accept incoming traffic in the MTD module without performing translation. This would expose the host on the non MTD link, potentially elimination the protection gained by using MTD. It would also be possible to allow all non-MTD connections by default and only translate matching connections. This would allow adversaries to discover the host and target it but could help hide certain connections and services that are only performed over MTD. Overall, either change greatly reduces the protection granted by the MTD scheme.

More general techniques to reduce overhead would be to disable certain address layers, reducing the amount of processing. Additionally, weaker cryptography can be employed. A smaller key can be used for HMAC, and a weaker HMAC function could be used, such as SHA1. The context data size could also be made smaller to reduce digest processing computation. If TLS is not being used, the TinyCrypt library could be used in place of MbedTLS for performing HMAC, which would reduce the Zephyr image code size. Finally, address rotation can be done infrequently to save on the cost of rotation, but at the cost of allowing an adversary more time to analyze connections.

5.7 Supporting Additional Protocols

The current implementation handles IPv4, UDP, TCP, and Ethernet, but additional protocols can be added to expand the existing framework. Branching paths can be introduced to detect the appropriate protocol at each layer, like the existing code that determines if TCP or UDP is being used. Additional code for handling IPv6 can be added to properly read and write the IPv6 header and translate IPv6 address structures using the existing IPv6 network stack code for reference. This greatly increases the address space available for mutation if the other devices in the network support IPv6. Protocol branching code can also be provided at the link layer to Bluetooth addresses similarly to Ethernet addresses. With Bluetooth, link keys can be pre-placed and make use of Bluetooth link encryption and authentication instead of using TLS if sufficient to meet requirements. Zephyr also supports 802.15.4 with 6LoWPAN which has significant differences in its stack and thus will require additional work. 802.15.4 MAC addresses can be handled like Ethernet addresses with the distinction of needing to support both standard EUI-64 addresses and short addresses. Special handling for the IPv6 address and port, which are likely to be in a compressed header will need to be added to handle the 6LoWPAN layer. Header compression will be limited based on the address space size used for random IPv6 addresses, and tradeoffs will have to be made between address length and reduced header sizes based on the system integrators use case. Appropriate handling of mesh address headers is needed if mesh routing is going to be implemented as part of the scheme. 802.15.4 link encryption and authentication can be used with pre-placed keys in place of TLS if sufficient to meet requirements. Wi-Fi addresses can also be handled similarly to Ethernet and WPA2 can be used for authentication and encryption in place of TLS. A related area of interest could be to randomize service set identifiers in the scheme to hide Wi-Fi infrastructure from adversaries.

The current implementation is designed for use with application layer sockets and has not been integrated into network libraries that provide high level APIs that abstract away socket handling. Use of MTD can be made more convenient to users if implemented as part of these libraries. As an example, the MQTT library could be updated with conditional compilation features to use MTD sockets and provide function calls for the user to provide keys and context data for connections to the MQTT library and for the MQTT library to rotate addresses. In general, many of the suggested solutions, improvements, and options discussed throughout section 5. of this paper could be implemented in separate libraries or as conditional features to allow system developers and integrators to choose their desired scheme implementation.

5.8 Comparison to Other Approaches

The implemented scheme builds on the foundation of NMTD schemes but focuses on integrating the highest security approaches from each and providing host-centric implementation. The scheme builds heavily on the concept of deriving addresses from shared keys introduced by MT6D. Unlike MT6D, the scheme has reduced network traffic overhead by directly using the derived addresses rather than including them on top of tunneled traffic. Based on the deployment environment, it is not assumed that a central controller or DNS server may be available, differing significantly from SDN based schemes and customer gateway schemes that centrally manage addresses. The scheme makes use of unique identifiers for every connection rather than using single identifiers per host, making it appear as if more hosts are active and making it difficult to resolve multiple sets of identifiers to a single host. Further, all common layers of identifiers are included in the scheme to maximize the address space and create additional uncertainty in resolving a host’s identity. An unchanging part of the identifier cannot be used to determine a unique host handling multiple connections. Keys are also handled per host or per connection (depending on if the application reuses the key for multiple connections with the same host, but different context data) rather than sharing a single key across an entire network. This introduces additional management overhead but ensures each connection is uniquely authorized, and that the compromise of a single host does not compromise the entire network.

CHAPTER 6. CONCLUSION

Implementing moving target defensed in embedded systems provides adaptive protection om increasingly networked. special-purpose systems that warrant additional security when deployed in sensitive contexts. Implementing a high security shuffling technique in the Zephyr RTOS revealed design and implementation challenges and solutions. The Zephyr network stack which had to be modified to implement the scheme due to the lack of rich user APIs and libraries present, a common theme when comparing RTOS to enterprise operating systems. The implemented scheme makes use of identifiers at all layers to maximize attacker uncertainty and avoid static identifiers that can be used to correlate connections. Using per connection identifiers and per host keys reveals a minimal amount of information to other network participants and limits the impact of host compromise. Incorporating existing support for transport layer security in Zephyr greatly enhances the secure provided by the scheme by further limiting spoofing attacks and hiding data that can be used to track connections over rotation periods. The access control and identity protection provided by the scheme greatly increases the work required of adversaries to carry out successful attacks.

The current implementation is a working proof of concept that has limitations in terms of data forwarding and collision handling. Approaches for handling forwarding and address collisions using either distributed protocols involving forwarding devices and hosts or introducing gateways provides insight into future work to adapt this scheme for wider use. The scheme’s implementation is intended to be flexible to allow adaptation to different use cases including reducing overhead and introducing additional protocols. The work of this paper helps system designers deal with implementation challenges in RTOS systems and provides them with a framework for adopting MTD in Zephyr.

REFERENCES

[1] J. Cho *et al.*, "Toward Proactive, Adaptive Defense: A Survey on Moving Target Defense," in *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 709-745, Firstquarter 2020.

[2] V. Heydari, "Moving Target Defense for Avionic Systems," *2018 National Cyber Summit (NCS)*, Huntsville, AL, pp. 53-57, 2018.

[3] L. Yue-Bin *et al*., “RPAH: A Moving Target Network Defense Mechanism Naturally Resists Reconnaissances and Attacks,” *IEICE Transactions on Information and Systems*, vol. E100.D, no. 3, pp. 496–510, 2017.

[4] C. Dishington, D. P. Sharma, D. S. Kim, J. Cho, T. J. Moore and F. F. Nelson, "Security and Performance Assessment of IP Multiplexing Moving Target Defence in Software Defined Networks," *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, Rotorua, New Zealand, 2019, pp. 288-295.

[5] M. Dunlop, S. Groat, W. Urbanski, R. Marchany and J. Tront, "MT6D: A Moving Target IPv6 Defense," *2011 - MILCOM 2011 Military Communications Conference*, Baltimore, MD, 2011, pp. 1321-1326.

[6] D. P. Sharma, J. Cho, T. J. Moore, F. F. Nelson, H. Lim and D. S. Kim, "Random Host and Service Multiplexing for Moving Target Defense in Software-Defined Networks," *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, Shanghai, China, 2019, pp. 1-6.

[7] D. P. Sharma, D. S. Kim, S. Yoon, H. Lim, J. Cho and T. J. Moore, "FRVM: Flexible Random Virtual IP Multiplexing in Software-Defined Networks," *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, New York, NY, 2018, pp. 579-587.

[8] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. 2012. “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* (*HotSDN ’12*). Association for Computing Machinery, New York, NY, USA, 127–132. DOI:https://doi.org/10.1145/2342441.2342467

[9] Douglas C. MacFarland and Craig A. Shue. 2015. “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* (*MTD ’15*). Association for Computing Machinery, New York, NY, USA, 37–41. DOI:https://doi.org/10.1145/2808475.2808485

[10] Michal Piskozub, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. 2019. “On the Resilience of Network-based Moving Target Defense Techniques Against Host Profiling Attacks,” In *Proceedings of the 6th ACM Workshop on Moving Target Defense* (*MTD’19*). Association for Computing Machinery, New York, NY, USA, 1–12. DOI:https://doi.org/10.1145/3338468.3356825

[11] Marc Green, Douglas C. MacFarland, Doran R. Smestad, and Craig A. Shue. 2015. “Characterizing Network-Based Moving Target Defenses,” In *Proceedings of the Second ACM Workshop on Moving Target Defense* (*MTD ’15*). Association for Computing Machinery, New York, NY, USA, 31–35. DOI:https://doi.org/10.1145/2808475.2808484

[12] Richard Skowyra, Kevin Bauer, Veer Dedhia, and Hamed Okhravi. 2016. “Have No PHEAR: Networks Without Identifiers,” In *Proceedings of the 2016 ACM Workshop on Moving Target Defense* (*MTD ’16*). Association for Computing Machinery, New York, NY, USA, 3–14. DOI:https://doi.org/10.1145/2995272.2995276

[13] Jason Li, Justin Yackoski, and Nicholas Evancich. 2016. “Moving Target Defense: a Journey from Idea to Product,” In *Proceedings of the 2016 ACM Workshop on Moving Target Defense* (*MTD ’16*). Association for Computing Machinery, New York, NY, USA, 69–79. DOI:https://doi.org/10.1145/2995272.2995286

[14] K. Zeitz, M. Cantrell, R. Marchany and J. Tront, "Designing a Micro-moving Target IPv6 Defense for the Internet of Things," *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, Pittsburgh, PA, 2017, pp. 179-184.

[15] F. Nizzi, T. Pecorella, F. Esposito, L. Pierucci and R. Fantacci, "IoT Security via Address Shuffling: The Easy Way," in *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 3764-3774, April 2019.

[16] K. Zeitz, M. Cantrell, R. Marchany and J. Tront, "Changing the Game: A Micro Moving Target IPv6 Defense for the Internet of Things," in *IEEE Wireless Communications Letters*, vol. 7, no. 4, pp. 578-581, Aug. 2018.

[17] M. Albanese, A. De Benedictis, S. Jajodia and Kun Sun, "A moving target defense mechanism for MANETs based on identity virtualization," *2013 IEEE Conference on Communications and Network Security (CNS)*, National Harbor, MD, 2013, pp. 278-286.

[18] Matthew Sherburne, Randy Marchany, and Joseph Tront. 2014. “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* (*CISR ’14*). Association for Computing Machinery, New York, NY, USA, 37–40. DOI:https://doi.org/10.1145/2602087.2602107

[19] Ward, Bryan C., et al. *Survey of Cyber Moving Targets Second Edition*. No. TR-1228. MIT Lincoln Laboratory Lexington United States, 2018.

[20] S. Sengupta, A. Chowdhary, A. Sabur, A. Alshamrani, D. Huang and S. Kambhampati, "A Survey of Moving Target Defenses for Network Security," in *IEEE Communications Surveys & Tutorials*.

[21] Cai, Gui-lin, et al. "Moving target defense: state of the art and characteristics." *Frontiers of Information Technology & Electronic Engineering* 17.11 (2016): 1122-1153.

[22] Zephyr Project members and individual contributors. “Network Stack Architecture.” zepyhrproject. https://docs.zephyrproject.org/1.14.1/guides/networking/net-stack-architecture.html.

[23] Zephyr Project members and individual contributors. “Zephyr Project Documentation.” zepyhrproject. https://docs.zephyrproject.org/1.14.1/.

[24] Zephyr Project. “Zephyr Project.” zepyhrproject. https://zephyrproject.org/.