International Conference on Information Society (i-Society 2014)

Implementing Dynamic Address Changes in ContikiOS

Tanner Preiss, Matthew Sherburne, Randy Marchany, Joseph Tront
Bradley Department of Electrical Engineering and Computer Engineering
Virginia Tech Information Technology Security Office
Virginia Tech, Blacksburg, Virginia 24061
Email: {tpreiss, msherbur, marchany, jgtront}@vt.edu

***Abstract*—To secure the Internet of Things (IoT) running on IPv6 over Low-Powered Wireless Personal Area Networks (6LoWPAN) by implementing a Moving Target IPv6 Defense (MT6D), there must first be a method developed to dynamically change IPv6 addresses over this resource constrained wireless network. We will discuss how we implemented this dynamic address change and how in doing so, we also had to change the Rime and MAC address thus implementing a moving target defense at Layers 2 and 3 of the OSI Model. We conducted our experiments in Contiki OS and utilized the Routing over Low-powered and Lossy Networks (RPL) protocol to perform the address changes between a node and its parent border router.**

***Index Terms*—6LoWPAN, MT6D, IPv6, RPL, Contik OS, IoT**

I. Introduction

The devices that make up the IoT, such as motion, temperature, and power sensors can be directly accessible over the Internet now that these devices are IPv6 addressable. A malicious actor has the ability to monitor traffic originating from a static IP address. There needs to be a method in which to implement a moving target defense on these low-powered, resource-constrained devices.

We will discuss the background of research into the security of IPv6, Rime stack, μIP, MT6D, 6LoWPAN, RPL, and ICMPv6 RPL Control Messages in Section 2. In Section 3 we will discuss a typical 6LoWPAN design and our 6LoWPAN test bed design. Section 4 discusses the initial address setup in Contiki OS. Section 5 will discuss our current implementation to dynamically assign changing IPv6 addresses to wireless sensor nodes. Section 6 will discuss our observations of the effect that changing addresses has on the network. Finally, Section 7 and 8 presents our Future Work and Conclusion.

II. Background

To understand how to dynamically change addresses in a wireless sensor that is running Contiki OS [1], we must first discuss the network protocols, network stacks, and control messages that are used in Contiki OS and then discuss how a global IPv6 address is formed in Contiki OS. We will then discuss in detail the implementation to dynamically change an IPv6 global address in Contiki OS, in the application layer.

Contiki OS prides itself on the implementation of IPv6 in a wireless sensor network, however Contiki OS accomplishes this by means of compression and using a combination of protocols to communicate to the global Internet and to local nodes.

A. Contiki OS Challenges

We preface this discussion by identifying a few inherent challenges and difficulties that Contiki OS contains, as do other operating systems which predominantly reside on embedded systems such as TinyOS[2], BeRTOS[3], SymbOS[4]. The first and most important challenge is that Contiki OS is designed to compress all addresses and retain only a few addresses that are able to form larger IPv6 addresses. Although the Contiki OS is written in the C language, the limited resources on embedded systems mean that traditional libraries and data structures are not included. This includes the common data structures used in IPv6 and networking code. The second challenge is that the entire Contiki OS is compiled and uploaded to the embedded system at once. This means that all the applications are loaded at compile time and executed using a system of timers. The limited ROM space on embedded systems means that each byte of memory is valuable and so ContikiOS has addressed this issue by allowing access to all memory at all times, meaning that no divisions exist between kernel and application level code. This can be valuable but also means that every global variable has the potential to be accessed from various locations throughout the code. Complete knowledge of the codebase and system design is necessary when dealing with global variables as the amount of resources necessary for a lock or mutex is often too large to be feasible.

B. IPv6

IP version 6 (IPv6) [5] was developed in response to the shortage of IPv4 addresses. IPv6, with 128-bits of addressing, allows for significantly more devices to be uniquely addressed on the Internet than with IPv4s 32-bits of addressing. The large address space provides ample addressing for the expansive Internet of Things.

C. Moving Target IPv6 Defense

Virginia Tech researchers, M. Dunlop et. al [6], developed an IPv6 defense scheme that provides security through obfuscation by rapidly changing addresses on two end points, similar to frequency hopping in radio communications. We

978-1-908320-38/4/$25.00©2014 IEEE

Authorized licensed use limited to: to IEEExplore provided by University Libraries | Virginia Tech. Downloaded on April 06,2023 at 19:07:05 UTC from IEEE Xplore. Restrictions apply.

are using this defense mechanism as a future goal for wireless sensor networks.

D. 6LoWPAN

IPv6 cannot be used to provide an address to just any device. The majority of wireless sensor networks utilize IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) in order to adapt IPv6 on top of the wireless sensor network protocol. RFC 4919 [7] and RFC 4944 [8] address the issues of assigning IPv6 to wireless sensor network devices. The base specification of 6LoWPAN is RFC 6282 [9] and a Neighbor Discovery update to the protocol is RFC 6775 [10].

E. Rime Stack and μIP (micro IP)

Directly related to IPv6 is Contiki OS' version of the IPv6 stack called the Rime stack [11]. This stack uses Rime addresses, and is used when the cost of a full IPv6 is too large for the platform's constrained resources. The Rime stack is capable of multicast communications, node to node communications, as well as more complex situations such as thwarting network floods. The Rime stack operates to minimize the bandwidth usage as well as minimize power consumption on routers and nodes where sleep cycles are implemented.

μIP is an open source TCP/IP stack that was originally developed by Adam Dunkels, also the creator of Contiki OS. In 2008, Cisco adopted the μIP stack and created a μIPv6 stack to accompany the original implementation. The μIP stack is originally designed for use on platforms that do not have operating systems and provides a stack that uses very little code, overhead, and processing power to maintain. It primarily accomplishes this compression by only using a single packet buffer instead of the typical IP protocol stack which stores many copies of packets.

F. RPL

RPL is IPv6 Routing Protocol for Low-Power and Lossy Networks and is defined in RFC 6550 [12]. For the purpose of our test bed design we will define the primary terms for RPL network design as being RPL Instance, Directed Acyclic Graph (DAG), Destination-Oriented Directed Acyclic Graph (DODAG) and then define the ICMPv6 RPL control messages: DODAG Information Object (DIO), DODAG Information Solicitation (DIS), Destination Advertisement Object.

RPL uses a hierarchy of terms to group the nodes into a network topology. In brief, a single Low-Power and Lossy Network (LLN) consists of one or more RPL Instances, an RPL Instance consists of one or more DODAGs each with the same RPLInstanceID, and finally a DODAG consists of one or more RPL nodes where at least one node is the DAG root.

RPL Instances function independently of other RPL Instances and this means that an RPL node can be both a leaf node in one instance and a root node in another instance. In our test bed, described in section 3, we have a single DODAG contained within a single RPL Instance, but it is easy to see how the node relationships and neighborhoods can become complicated in an enterprise setup. A DAG is a directed acyclic graph, where all edges are oriented such that no cycles exist and all edges terminate at one or more root nodes. In our setup, a DODAG is a DAG, but with a single DAG root, such that all edges are directed and terminate at the DAG root. Furthermore our DAG root acts as a border router for the DODAG and aggregates routes in the DODAG which are used to communicate with other protocols such as IPv6.

G. ICMPv6 RPL Control Messages

After understanding the structure of a LLN, we need to understand how the communication occurs within an RPL Instance. This is made possible by RPL control messages [11] which are a new ICMPv6 message. ICMPv6 messages were originally specified in RFC 4443 [13].

The three main control messages that pertain to our implementation are DODAG Information Object (DIO), DODAG Information Solicitation (DIS), and Destination Advertisement Object (DAO). A DIO control message carries information that allows an RPL node to discover a RPL Instance, learn the Instances configuration parameters, select a DODAG parent set, and maintain the DODAG. These messages are constantly being sent from RPL nodes at a code specified minimum interval. For more information on changing the interval refer to rpl-conf.h, which defines a variable RPL_DIO_INTERVAL to default 12. Where $n = 12$ and the minimum interval is defined as 2^n milliseconds, so, 2^{12} ms is 4.096 seconds. We mention this here because DIOs are the most common control message and in section 6, we mention the results of changing the RPL_DIO_INTERVAL value to increase DIO packets.

The second control message, DIS, is used to solicit a DODAG Information Object from an RPL node. A DIS is analogous to a Router Solicitation in IPv6 Neighbor Discovery, a node may use DIS to probe its neighborhood for nearby DODAGs. In the analysis section, we will discuss the necessity of a DIO/DAO pair of messages to maintain and establish a global route.

The final control message, DAO, is used to propagate destination information upward along the DODAG and establish downward routes towards the DAG root. This message is essential for route addition into the DAG root's route table.

Summarizing the workings of RPL control messages, some nodes are configured as a DAG root with associated DODAG configurations. Nodes advertise initial presence by first sending a DIS message which is responded to by DIOs of neighboring nodes. The nodes are constantly listening for and transmitting DIOs and use the incoming DIOs information to join new DODAGs. Once a node joins a DODAG, it needs a DAO to determine and establish itself in the DAG root

III. 6LoWPAN Design

A. Typical Design

Typical 6LoWPAN network designs include a border router actively monitoring and responding with no duty cycle. This border router is usually attached to an exterior power source and/or greater processing power such as a laptop. This larger processing power has the capability to connect to the global

Authorized licensed use limited to: to IEEExplore provided by University Libraries | Virginia Tech. Downloaded on April 06,2023 at 19:07:05 UTC from IEEE Xplore. Restrictions apply.

Internet and uses the USB connected mote to bridge the global Internet with the 6LoWPAN network. A series of motes then connects to the border router and use the 802.15.4 protocol to establish communication across a local 6LoWPAN network.

B. Test Bed Design

Our implementation was initially designed by Sherburne et al [14] and utilized wireless sensor motes. Our current test bed design, see Figure 1, extends this original design by adding a wireless sniffer mote to analyze 6LoWPAN traffic.

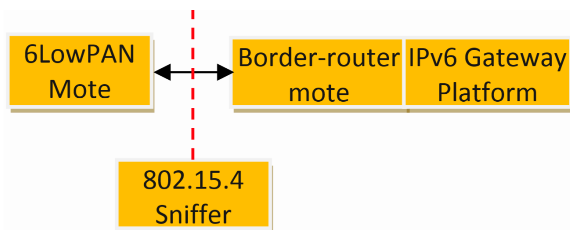

Fig. 1. 6LoWPAN Test Bed Design

There are two options for implementing MT6D on a wireless sensor network. The first option requires MT6D operating on the border router because the border router is the only gateway that a 6LoWPAN mote has to the Internet. The first option offers a simpler design because our border router runs on a Raspberry Pi and MT6D has been shown to run on a non-embedded Linux platform. Our second option implements MT6D as originally intended, on end-to- end communication points. We selected option two because we recognize that in an enterprise network or the proposed Smart Grid ideas [15], control of the border router is not always guaranteed and MT6D on the 6LoWPAN mote ensures that an attacker cannot effectively sniff the IPv6 traffic to the mote. In our test bed design, we implement a sniffer to analyze network activity.

1) Hardware and Network Design: The 6LoWPAN low-power wireless sensors for this design are the Tmote Sky motes [16]. Although relatively under-powered with a 8 MHz TI MSP430 microcontroller, 10kB of RAM, 48kB of ROM, and a TI CC2420 IEEE 802.15.4 compliant radio, it serves as a resource-constrained environment in which to validate the efficacy of dynamic address changing in 6LoWPAN. We established one Tmote Sky as an RPL Border Router, one Tmote Sky running a udp-echo-server, and one Tmote Sky running as a network sniffer interface for Foren6 [17], a program used to analyze 6LoWPAN traffic. We used version 2.7 of Contiki OS on the Tmote Sky running RPL Border Router and another Sky running UDP Echo Server.

2) RPL Border Router: We utilized the same RPL Border Router setup as in Sherburne et al [14] with a Raspberry Pi and Tmote Sky.

3) UDP Echo Server: Our second Tmote Sky runs with the Contiki example program udp-echo-server. This program allowed us to establish minimal code overhead and simulate basic traffic over IPv6 to the Tmote Sky and other platforms capable of UDP communications. Upon boot-up, these Tmote Skys attach a portion of their MAC address to the IPv6 prefix to form their Global-Link address. This method has security concerns because an attacker could find out the MAC address of a device and try to correlate that with the Global-Link address. This provides yet another motivation to change IPv6 addresses to protect against IP address to physical device correlation.

4) Network Sniffer: Our third Tmote Sky runs a Contiki sniffer [18] example that combined with a program called Foren6 [17] is able to passively capture 6LoWPAN traffic and render the network state in a graphical user interface. Both Foren6 and the Contiki sniffer [18] example are developed by the Center of Excellence in Information Technology and Communication (CETIC), a Belgian ICT applied research center that provides expertise in embedded, cloud and service oriented technologies, and the code can be found on CETICs github page. We use the TMote Sky sniffer as an interface to Foren6 which provides us with event data that can be further broken into packet data and further is read and graphically displayed as nodes with arrows associating their parents/neighbors.

IV. Contiki OS Initial Setup

Now that we have a notion of the different protocols and the Rime stack used in Contiki OS, we describe how the Contiki OS performs initial setup and address configuration. We are describing the initial setup so that the solution, described in section 5, will be validated with respect to ordering and setting of global variables. Each of the conversions and addresses that are used in the initial setup have to be replicated in our implementation to ensure that all structures and addresses are changed to the new address.

The code in contiki-sky-main.c starts by applying a conversion to a global array labeled ds2411_id. The conversion changes the 802.15.4 MAC address into a compatible EUI-64 bit address [19]. This variable is the base of all addresses created. Upon initialization ds2411_id contains a unique value that derives from the hardware specifications including family, type, and node_id. The ds2411_id is a unique identifier that is used as the MAC address of the node. It is important to note that we opted not to change the ds2411_id data structure, as we felt that a Layer 7 application should not be changing hardware specific unique identifiers.

Shortly after the ds2411_id conversion is performed; set_rime_addr(), a local method, performs a memcopy of the ds2411_id and finalizes the Rime address using rimeaddr_set_node_addr(&addr). As mentioned before, the Rime stack provides a specialized network stack when the full IPv6 stack is too large. This is the first method that relies on the ds2411_id, and although we are not modifying the ds2411_id directly, we need to ensure that the address, which we create dynamically, meets the required Ethernet MAC address specification.

The next part is the cc2420_init() local method. This method initializes the CC2420 radio onboard the TMote Sky platform.

Authorized licensed use limited to: to IEEExplore provided by University Libraries | Virginia Tech. Downloaded on April 06,2023 at 19:07:05 UTC from IEEE Xplore. Restrictions apply.

The method sets the radios PAN (Personal Area Network) address to the Rime address which is set immediately before this method that is derived from the ds2411_id. This step is included because the CC2420 radio acknowledges all inbound packets' destination addresses via cross-checking its PAN address with the destination of the packet. There is a way to put the radio into promiscuous mode and accept all incoming traffic but this is not useful for our implementation as this would break the security we wish to obtain through MT6D.

The setup continues onto another memcopy of the ds2411_id into the global variable: uip_lladdr.addr. The uip_lladdr.addr variable is the link local address that all of the IP stack will use during execution. On a side note, all link local IPv6 addresses on the TMote Sky platform running Contiki OS, begin with the address fe80. As we mentioned before, we are not changing the ds2411_id, instead we will use this initialization information to create an implementation that mimics this behavior and dynamically changes the address via a timer.

The final part in the setup is the most crucial and under documented section of the code, and it provides the backbone of our implementation detailed in the next section. To begin, once we have set the Rime address, uip_lladdr.addr, and the CC2420 radio PAN address, the TMote Sky completes initialization of other systems such as the temperature sensors and onboard LEDs. However, the code does not explicitly note that there is a global array called uip_ds6_if, with a default size of 3, which stores the addresses which are used as the global and link local address. Upon initialization of the above addresses, the uip_lladdr.addr is added to this array as a link local address. This address is used as the source of the first DIS control message which is sent upon initialization of the TMote Sky node. As mentioned before the RPL node will now receive a DIO control message from neighboring nodes containing information about the surrounding DODAG as well as the prefix to which a global address can be created. The node now adds this prefix to a global array of prefixes, and then uses the newly obtained prefix to create a full global address by combining the prefix with its link local address that it derived from the IP link local address. The node finally adds this global address into the global list of addresses: uip_ds6_if. At this point the node has now established two addresses in the global address array, both a link local and its global equivalent, complete with prefix from the DIO control message.

V. Implementation

We have detailed the initialization of the global address in the previous section. Our goal now is to replicate this behavior in a single application layer method that can accept an IPv6 address as a parameter and set the nodes global address to the input address. We should note that our implementation has worked in the testing that we have performed, however, this implementation should only be used as a proof of concept and surely has room for improvement.

We demonstrate a simplified dynamic address change by incrementing the last octet of the global IPv6 address by 1 every 1-10 seconds. We then proceed to call our method which sets the global address to this new value. Our method takes, as a parameter, the lower 64 bits of an IPv6 address, via eight 8-bit unsigned integers, and we have hardcoded the address prefix for use in the global address. The hardcoded prefix does not seem far from the final implementation as it is not uncommon to specify a subnet in a configuration file. Also note that MT6D uses these lower 64 bits to obfuscate and create a new address. Our first part of the implementation includes the conversion to make the address Ethernet compliant. We bitwise XOR bits 40-47 with 0xfe which performs the conversion and flips the 7th bit to 1 to be Ethernet compliant.

We then remove the old local and global addresses from the global array mentioned in the previous section that stores the addresses for the node. In an effort to save code on the embedded system, we explicitly modify the global array which stores the addresses for the node and mark the addresses as not used, replicating the behavior of uip_ds6_addr_rm. These array positions will be overwritten upon a following add into the array. We noted that if the addresses are not explicitly removed, adding into the global array will fail rather than overwrite, and the address will not be added.

Now that we have ensured that the incoming address is Ethernet compliant and have established free space for the new address, we can begin to add a new address into the uip_net_if list and set the global variables mentioned in the previous section. To add the addresses, we must first create a uip_ipaddr_t struct for both the local and global addresses and then initialize them using uip_ip6addr_u8() method which takes as a parameter: a 128-bit number broken into sixteen 8-bit numbers. We then use uip_ds6_addr_add(), this takes as parameters: the IP address, lifetime, and address type. We have the lifetime set at 0, which implies that the address lifetime is infinite. In a future implemention of MT6D on a Tmote Sky we may set these lifetimes to a specified interval. We have experimented slightly with the address type, which can be ADDR_MANUAL, ADDR_TENTATIVE, or ADDR_PREFERRED, and have decided to use ADDR_PREFERRED as this is the type that is used upon initial setup of an address and also provides more precedence when performing duplicate address detection.

Now that we have successfully removed an old address and added a new address, we need to perform the bookkeeping of global variables mentioned in the Contiki setup section above. We have to set the Rime address, CC2420 PAN address, and global uip_lladdr.addr. This ensures that any address that will be used throughout the messaging and stack protocols will be updated to the new address. We first create a 64-bit Rime address struct rimeaddr_t named addr and set its values by accessing the address as an 8-bit array to perform these address changes. This is accomplished by accessing the rimeaddr as addr.u8[i] where i is the index, from 0-7, to change this dynamic Rime address to the new address.

Similar to the contiki-sky-main.c setup we are able to set the Rime address using rimeaddr_set_node_addr(). We set the CC2420 PAN address by copying the newly set Rime address and using cc2420_set_pan_addr. In our last bookkeeping of

978-1-908320-38/4/$25.00©2014 IEEE

Authorized licensed use limited to: to IEEExplore provided by University Libraries | Virginia Tech. Downloaded on April 06,2023 at 19:07:05 UTC from IEEE Xplore. Restrictions apply.

global variables the uip_lladdr.addr is setup by again copying the newly set Rime address into this structure using memcopy.

Our final, critical section, is sending a DIS, a DIO, and a DAO. The DIS control message destination is the multicast address ff02::1a. As described before, this will begin the cycle of control messages that will eventually add the new address as a globally addressable route to the border router.

In order to send a DIO and DAO message we must use dio_output() and dao_output() respectively, but these methods both take an RPL Instance as a parameter. Our implementation loops over the global instance_table and sends a DIO and DAO to each of the RPL Instances. In our test bed design, our node is only associated with one RPL Instance and so this loop is only executing once for our base case.

To summarize our implementation: 1. Perform conversion on new address to ensure Ethernet compliant. 2. Remove old link local and global address from uip_net_if list. 3. Add new local and global address to uip_net_if list. 4. Set Rime address 5. Set CC2420 address 6. Set global uip_lladdr address used in μIP stack. 7. Send a DIS, DIO, and DAO control message.

VI. Analysis

Using our test bed design we were able to capture live traffic with Foren6. This data provided most of our analysis coupled with the web server running on the border router which displays an HTML page. The HTML page details the border routers neighbor table and route table. Using the displayed routes, a user is able to access each specific route in a browser which provides a new HTML page that displays readings from the temperature sensor.

The goal of this paper is to prove that dynamic address changes can occur in Contiki OS; however, the goal of our experimentation is to find the address change time interval on a mote to which the border router is able to achieve a near 100% route addition success rate. This experiment is also crucial for our future work with MT6D on an embedded device.

In order to create data points for measuring the amount of addresses that are properly added into the border router, we first found in the Contiki OS source code that a mote stores many tables including a neighbor table and a route table. We implemented a method that would print when a node was added to the neighbor table of the border router and print again when the same node was added to the border routers route table. We kept a running total of both the neighbor table and the route table which tracked the total neighbors and total routes that were added. This way we were able to quantify the total number of address changes and total number of sucessful address changes, and furthermore maintain a list of the addresses that were successful. We define a successful address change as an address that has a route established in the border router.

Once we established that we would use Foren6 to capture live data and the border router printf to establish which routes were successful, we saved both of these outputs in log files and began experiments.

Our experiment consisted of changing addresses at a set interval. This interval was changed from 1 to 10 seconds in 1 second increments and at each time interval, we ran the experiment 10 iterations, for a total of 100 tests. In each test we wanted 100 successful address changes but we collected 125 successful changes. We chose 125 address changes for two reasons. First, the border router has a maximum table size of 20, due to memory size constraints, after the maximum size; a replacement policy is in place and we want all addresses to be added under replacement policy to ensure that time for adding is the same and to closely resemble an enterprise border router with rotating route tables. Our second reason for 125 addresses is an additional 5 address changes to account for any errors. The data now becomes 100 successful address changes with 125 total successful address changes collected minus 20 (the first 20 dropped because they are added with without replacement policy) minus 5 (inconsistent errors) = 100 successful address changes.

We found that there are addresses added to the router's neighbor table which are not always added to the router's route table. We also found a slight inconsistency of number of addresses added to the neighbor table. For this reason we created a ratio using the Foren6 data, we use the total address changes witnessed by Foren6 and the total successful routes (100 in all tests) and create a ratio to determine success rate of each interval for changing addresses. In Figure 2 we show a graph of the 100 tests with success rate vs address change interval.

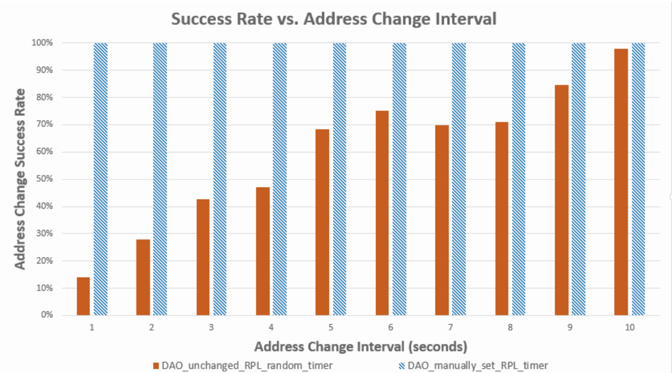

Fig. 2. Route Addition Success Rate

At a 10 second address change interval, a 100 percent route addition success rate was achieved. This means that a DAO frame was sent for every address change and that frame was also received by the border router and a route established. This can be explained by the time delay between when a DIS frame, which initiates the address change, is sent to when the DAO frame is sent which advertises the global link address to the border router. This delay was observed as high as 9.80 seconds.

We found that rpl-timers.c within Contiki OS controlled dao_output() by a random timer in order to reduce the probability that any two nodes will transmit a DAO packet at the same time. In Figure 2, the data labeled

Authorized licensed use limited to: to IEEExplore provided by University Libraries | Virginia Tech. Downloaded on April 06,2023 at 19:07:05 UTC from IEEE Xplore. Restrictions apply.

”DAO_unchanged_RPL_random_timer” displays our test results after executing with the native random timer in Contiki OS.

In order to fully implement MT6D, we must be able to advertise the next address to the border router as quickly as possible to still allow time for sensor data to be sent within that address interval. We had to manually call dao_output() and disable the time process in rpl-timers.c thus adapting the RPL protocol in order to allow rapid transmission of control messages. With this change, we were able to transmit the DAO packet immediately after the DIS. We found the Tmote Sky running as the border router was not able to fully process the incoming DIS and DAO packets fast enough. A 0.30 second delay was added between transmission of the DIS and DAO packets. Following these changes, we repeated the same experimental setup as discussed before and achieved between 99% and 100% route addition success rates. The experiment data illustrating our manual changes is represented in Figure 2 by ”DAO_manually_set_RPL_timer.”

VII. Future Work

We will continue to implement MT6D within Contiki OS by adding the Secure Hashing Algorithm (SHA) library in order to build randomized addresses. The code implementation presented consists of 45.27 KB that includes, the base Contiki OS, the base UDP echo server and dynamic address changing applications. This is almost at the Tmote Sky’s maximum memory size of 48 KB. In order to move past the memory constraints of the Tmote Sky, we will continue our research with another platform, the Econotag II [20], which provides 96KB of RAM.

VIII. Conclusion

We have shown that we can successfully change the MAC, Link-local and Global-link addresses of a wireless sensor at regular intervals as quickly as 1 second with a 99% route addition success rate at the border router. This method of dynamically changing addresses within 6LoWPAN provides an initial foundation on the efficacy of MT6D being implemented within wireless sensor networks.

We have presented that there is a need for applying a moving target defense to wireless sensors with IPv6 addresses in the Internet of Things in order to establish end-to-end security. Sensor data, no matter how inconsequential, can provide an attacker with critical information when aggregated over time or with data observed from multiple sensors. We showed in Section 4 how Contiki OS traditionally establishes addressing on a Tmote Sky. In Section 5 we described how we implemented dynamic address changes within Contiki OS on a Tmote Sky and then analyzed the results of successful address route additions in the border router in Section 6. This analysis concludes that we have a foundation in which we can implement the rest of the MT6D protocol including hashing addresses and establishing an encrypted tunnel.

References

[1] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki-a lightweight and flexible operating system for tiny networked sensors,” in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 2004, pp. 455–462.

[2] T. Alliance. (2012) TinyOS 2.1.2. [Online]. Available: hhttp://www.tinyos.net/ [Accessed: 2014-06-01]

[3] D. s.r.l. (2011) BertOS 2.7.0. [Online]. Available: http://www.bertos.org/ [Accessed: 2014-06-01]

[4] SymbiosiS. (2014) SymbOS 2.1. [Online]. Available: http://www.symbos.de/ [Accessed: 2014-06-01]

[5] S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification,” RFC 2460 (Draft Standard), Internet Engineering Task Force, Dec. 1998, updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112. [Online]. Available: http://www.ietf.org/rfc/rfc2460.txt

[6] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront, “Mt6d: A moving target ipv6 defense,” in *MILITARY COMMUNICATIONS CONFERENCE, 2011-MILCOM 2011*. IEEE, 2011, pp. 1321–1326.

[7] N. Kushalnagar, G. Montenegro, and C. Schumacher, “IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals,” RFC 4919 (Informational), Internet Engineering Task Force, Aug. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4919.txt

[8] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, “Transmission of IPv6 Packets over IEEE 802.15.4 Networks,” RFC 4944 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, updated by RFCs 6282, 6775. [Online]. Available: http://www.ietf.org/rfc/rfc4944.txt

[9] J. Hui and P. Thubert, “Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks,” RFC 6282 (Proposed Standard), Internet Engineering Task Force, Sep. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6282.txt

[10] Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann, “Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs),” RFC 6775 (Proposed Standard), Internet Engineering Task Force, Nov. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6775.txt

[11] A. Dunkels, “Rime-a lightweight layered communication stack for sensor networks.” 2007.

[12] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, “RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks,” RFC 6550 (Proposed Standard), Internet Engineering Task Force, Mar. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6550.txt

[13] A. Conta, S. Deering, and M. Gupta, “Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification,” RFC 4443 (Draft Standard), Internet Engineering Task Force, Mar. 2006, updated by RFC 4884. [Online]. Available: http://www.ietf.org/rfc/rfc4443.txt

[14] M. Sherburne, R. Marchany, and J. 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*. ACM, 2014, pp. 37–40.

[15] S. Groat, M. Dunlop, W. Urbanksi, R. Marchany, and J. Tront, “Using an ipv6 moving target defense to protect the smart grid,” in *Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES*. IEEE, 2012, pp. 1–7.

[16] J. Polastre, R. Szewczyk, and D. Culler, “Telos: enabling ultra-low power wireless research,” in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*. IEEE, 2005, pp. 364–369.

[17] S. Dawans and L. Deru. (2011) Troubleshooting with Foren6. [Online]. Available: https://github.com/cetic/foren6 [Accessed: 2014-06-01]

[18] S. Dawans. (2013) Sniffer 15.4. [Online]. Available: https://github.com/cetic/contiki/tree/sniffer/examples/sniffer [Accessed: 2014-06-01]

[19] R. Hinden and S. Deering, “IP Version 6 Addressing Architecture,” RFC 4291 (Draft Standard), Internet Engineering Task Force, Feb. 2006, updated by RFCs 5952, 6052, 7136, 7346, 7371. [Online]. Available: http://www.ietf.org/rfc/rfc4291.txt

[20] Redwire. (2014) Econotag II. [Online]. Available: http://redwire.myshopify.com/products/econotag-ii [Accessed: 2014-06-01]

Authorized licensed use limited to: to IEEExplore provided by University Libraries | Virginia Tech. Downloaded on April 06,2023 at 19:07:05 UTC from IEEE Xplore. Restrictions apply.